SDSC TR-1999-1

Evaluation of a Multithreaded Architecture for Defense Applications

Wayne Pfeiffer, Larry Carter, Allan Snavely,
Robert Leary, and Amit Majumdar
San Diego Supercomputer Center
University of California, San Diego

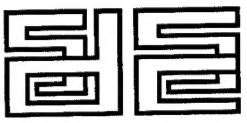
Sharon Brunett
California Institute of Technology

John Feo, Brian Koblenz, and L.G. Stern
Tera Computer

Joseph W. Manke
Boeing

Timothy P. Boggess
Sanders/Lockheed Martin

FINAL REPORT TO DARPA
September 1999



SAN DIEGO SUPERCOMPUTER CENTER
TECHNICAL REPORT

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	9/30/99	Final Technical Report (1 July 97-30 June 99)

4. TITLE AND SUBTITLE
Evaluation of Multithreaded Architecture for Defense Applications

5. FUNDING NUMBERS
DAR7-F188/00d

6. AUTHOR(S)
Wayne Pfeiffer, Larry Carter, Allan Snavely, Robert Leary, Amit Majumdar,
Sharon Brunett (Caltech), John Feo (Tera), Brian Koblenz (Tera), L.G. Stern (Tera)
Joseph W. Manke (Boeing), anad Timothy P. Boggess (Sanders/Lockheed Martin)

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
San Diego Supercomputer Center, University of California, San Diego
Center for Advanced Computing Research, Caltech, MC 158-79 Pasadena, CA 91125
Tera Computer Company, 411 First Avenue South, 600, Seattle WA 98104
The Boeing Company, Info. & Support Serv., P.O. Box 3707, Seattle, WA 98124
Sanders/Lockheed Martin, Adv. Engr Tech. Div, P.O. Box 868, Nashua, NH 03061

8. PERFORMING ORGANIZATION REPORT NUMBER
SDSC TR-1999-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA/ITO
3701 NORTH FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
DABT63-97-C-0028

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
DARPA/PM/ITO
DARPA/S&IO
DARPA/Technical Library
DTIC
ATZS-DKO-I

13. ABSTRACT (Maximum 200 words)

Multithreading has received considerable attention in recent years as a promising way to hide memory latency in high-performance computers, while providing access to a large and uniform shared memory. Tera Computer of Seattle has designed and built a state-of-the-art multithreaded computer called the MTA. Its intended benefits are high processor utilization, scalable performance on applications that are difficult to parallelize, and reduced programming effort.

The largest MTA and the only one outside of Seattle is at the San Diego Supercomputer Center (SDSC) on the campus of the University of California, San Diego (UCSD). Currently the MTA at SDSC has 8 processors.

The performance and usability of the MTA for 14 defense-relevant applications were evaluated in a two-year project described here. The applications included seven standard kernels, five mini-applications, and two large applications. The evaluation was led by researchers at UCSD with collaborators at Caltech, Tera, Boeing, and Sanders/Lockheed Martin. UCSD researchers also carried out multithreaded scheduler and compiler studies. The principal findings of the project follow in the enclosed final report.

14. SUBJECT TERMS	15. NUMBER OF PAGES
N/A	70
	16. PRICE CODE: N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unrestricted	Unrestricted	Unrestricted	None

SDSC TR-1999-1

Evaluation of a Multithreaded Architecture for Defense Applications

Wayne Pfeiffer, Larry Carter, Allan Snavely,
Robert Leary, and Amit Majumdar
San Diego Supercomputer Center
University of California, San Diego

Sharon Brunett
California Institute of Technology

John Feo, Brian Koblenz, and L.G. Stern
Tera Computer

Joseph W. Manke
Boeing

Timothy P. Boggess
Sanders/Lockheed Martin

Final Report to DARPA
September 1999

CONTENTS

SUMMARY

Multithreading has received considerable attention in recent years as a promising way to hide memory latency in high-performance computers, while providing access to a large and uniform shared memory. Tera Computer of Seattle has designed and built a state-of-the-art multithreaded computer called the MTA. Its intended benefits are high processor utilization, scalable performance on applications that are difficult to parallelize, and reduced programming effort.

The largest MTA and the only one outside of Seattle is at the San Diego Supercomputer Center (SDSC) on the campus of the University of California, San Diego (UCSD). Currently the MTA at SDSC has 8 processors.

The performance and usability of the MTA for 14 defense-relevant applications were evaluated in a two-year project described here. The applications included seven standard kernels, five mini-applications, and two large applications. The evaluation was led by researchers at UCSD with collaborators at Caltech, Tera, Boeing, and Sanders/Lockheed Martin. UCSD researchers also carried out multithreaded scheduler and compiler studies. The principal findings of the project follow.

MTA Performance

- For four applications tuned on the MTA and a reference high-end computer, the performance on 8 MTA processors is appreciably higher than on any number of processors of the reference computers. The MTA speed advantage ranges from a factor of 1.8 over a single vector processor for a large application to a factor of 5.4 over 12 workstation processors for a mini-application. For two other, kernel applications the speed on 8 MTA processors is comparable to the speed on 8 vector processors.

- For tuned applications, single-processor performance of the multithreaded Tera MTA (with a 260-MHz clock) is typically lower than that of the reference vector computer (with a 440-MHz clock). The vector processor is substantially faster than the MTA processor (by factors ranging from 2.2 to 6.5) for 5 out of 8 applications compared. The MTA processor is appreciably faster (by a factor of 1.5) for one application, an integer sort kernel.

- The MTA single-processor theoretical peak speed of three floating-point operations (two adds and one multiply) per clock cycle is not obtainable in practice. The best-achievable speed is two floating-point operations (one add and one multiply) per clock (i.e., 520 Mflops). This can be approached for dot products. By contrast the vector processor peak speed is four floating-point operations (two adds and two multiplies) per clock (i.e., 1,760 Mflops). This can be approached for well-vectorized linear algebra kernels.

- The speed of an MTA processor is also limited by its ability to issue at most one memory reference per clock cycle. Some common math kernels need to be rewritten to work around this limitation. By contrast the vector processor can issue four loads and two stores each clock.

- For tuned kernels the MTA consistently obtains a large fraction of its single-processor peak speed and appears to exhibit less variability in this fraction than does the vector processor. This suggests that multithreading has broader applicability than vectorization.

- For tuned applications, single-processor performance of the MTA is typically higher than that of cache-based, workstation processors of comparable clock speed. An MTA processor is substantially faster than a workstation processor (by factors ranging from 2.5 to 10) for 5 out of 6 applications compared. This indicates the effectiveness of multithreading as compared to cache utilization.

- Scalability on the MTA is appreciably better for the kernels studied than for the mini- and large applications. Indeed, 6 of the 7 kernels have good or very good scalability (parallel efficiency between 0.81 and 1.00 on 8 processors), while 5 of the 7 mini- and large applications have poor or fair scalability (efficiency between 0.50 and 0.71 on 8 processors). The larger applications have many

more sections of code that need to be tuned to achieve good performance, and insufficient time was available during the project for the extra tuning required.

- Typically, 40 or more streams are needed on each processor to hide memory latency and achieve full processor utilization. Since each hardware stream holds the context for a single software thread, this requires that the compiler and programmer find substantially more parallelism than on traditional parallel computers.

- Performance of serial sections of code is very slow on the MTA – slower than on a typical workstation. This arises because a stream can execute an instruction only once every 21 clocks. Tuning to minimize serial (single-threaded) code is thus critical for achieving high performance of an individual application.

- The overhead to acquire a stream is on the order of hundreds of instructions. This affects parallelization strategy and often makes performance on small problem sizes disappointing. In general, parallelizing a loop with few instructions and few iterations is counter-productive.

- The MTA is able to exploit inner-loop (vector) parallelism, but prefers outer-loop parallelism. The overhead to parallelize an outer loop is paid once, but the overhead to parallelize an inner loop is paid the number of times that the outer loop iterates (unless the loops are fused). Indeed, a good tuning strategy on the MTA is to first find parallelism and then move it to an outer loop.

- The MTA can synchronize at the word level with no additional cost over that of a normal memory reference. This capability enables parallel programming in a dataflow style when dependencies are complicated or not known until runtime. This allows some applications to be parallelized in ways not possible on other parallel computers.

MTA Usability

- The MTA's parallel programming model is very attractive. Multithreading is the only type of parallelization that need be considered, regardless of the number of processors.

- MTA usability also benefits from the lack of data caches. This eliminates the tuning for data locality that is typical on computers with workstation processors.

- Tera's sophisticated compilers embody impressive technology and automatically parallelize many constructs. In general, the Fortran compiler is more effective at parallelization than the C compiler. Nevertheless, neither compiler is able to achieve adequate multithreading automatically except for small kernels. Tuning, often by adding compiler directives, is thus essential for typical applications.

- Tuning is facilitated by good tools, especially Canal and Traceview. Canal annotates source code to show what the compiler parallelized, what it did not, and why. Traceview does a post-mortem analysis of an execution trace and makes a time-dependent plot of processor utilization and performance.

- Porting and tuning small applications to run on the MTA are roughly comparable to doing so for other uniform shared-memory computers and generally easier than for distributed-memory computers. One need not be concerned with data distribution across hierarchical memory.

- Porting and tuning large applications to run on the MTA have proved much more difficult. Because the MTA requires high levels of parallelism, more sections of serial code need to be parallelized than on other computers, and more parallelism must be found. This, coupled with the greater complexity of the applications, substantially increases the tuning effort.

- System instability, compiler bugs, compiler features (such as strict conformance to language conventions), and slow I/O have seriously reduced programmer productivity and lessened the inherent advantage of the MTA's parallel programming model. While software maturity improved during the course of the project, usage of the MTA at SDSC has been largely restricted to software development, evaluation, and computer science studies, rather than production computing.

Multithreaded Scheduling and Compilation

- Improved throughput has been demonstrated on the MTA when various combinations of kernel jobs are scheduled to run simultaneously on a single processor rather than sequentially. Such jobs are called "symbiotic." Through symbiotic job scheduling, parallel peaks in one job's execution profile can fill parallel valleys in another's profile.

- Highly tuned codes are less symbiotic than untuned versions of the same codes on the MTA. Because a parallel section cannot acquire streams released by another parallel section, highly tuned parallel codes with few serial sections can "throttle" each other and lead to reduced throughput. A more dynamic resource allocation strategy is being designed to address this issue.

- Tera's compilers often generate optimal code with regard to memory references. The generated code also uses an effective and low-overhead variant of dynamic scheduling to distribute iterations of a parallel loop to processors.

APPROACH AND FINDINGS

Objectives and Background

The principal objective of the two-year project described here was to determine conclusively whether a multithreaded computer architecture, embodied in the Tera MTA, would allow major defense applications to scale to sustained multi-gigaflops performance. Many such applications perform poorly or do not scale on conventional parallel computers despite extensive reprogramming. A further objective was to explore software approaches to multithreading that might broaden its applicability and improve performance over the current state of the art.

The achieved speed of conventional parallel computers on most applications is much lower than the peak speed. This occurs, in part, because data cannot be moved from memory to the processors fast enough to keep them fully busy. Moreover, the memory latency (i.e., time to retrieve data from memory) is not improving as fast as the processor clock period. Hence, management of memory latency is increasingly important to obtain high, sustained speeds. The most promising approach to manage memory latency in a scalable way is multithreading.

The Tera MTA, designed and built by Tera Computer of Seattle, is the state-of-the-art multithreaded computer investigated here. It has multithreaded processors and a high-bandwidth network to hide a large amount of latency to a large and uniform shared memory. Sophisticated compilers and performance tuning tools are available to extract parallelism from a broad spectrum of applications. Expected benefits of the MTA include high processor utilization, near linear scalability, and reduced programming effort.

The largest MTA and the only one operational outside of Seattle is at the San Diego Supercomputer Center (SDSC) on the campus of the University of California, San Diego (UCSD. The MTA at SDSC was installed in stages over the past two years, growing from one to two to four and most recently eight processors.

Acquisition of the 8-processor MTA was funded by NSF and DOE. Those agencies have also supported evaluations of the MTA for applications of interest to their user communities. The DARPA-funded project described here extended those evaluations to encompass defense applications.

Approach

The three technical tasks proposed and carried out were as follows:

- Implementation, optimization, and evaluation for the first time of defense applications on an operational multithreaded computer, the Tera MTA.
- Studies of the parallel profile of the applications and how that affects processor utilization and performance scaling to systems much larger than the modest-size MTA available at SDSC.
- Development and evaluation of compiler technology, runtime support, and adaptive techniques for multithreaded computers.

Significant progress was made on all three tasks with findings summarized in later sections and detailed in case study descriptions. Additional information on the approach used in each task follows here.

The work was led by researchers at UCSD with collaborators at Caltech, Tera, Boeing, and Sanders/Lockheed Martin. Biweekly conference calls were held throughout the two-year project to coordinate activities. In addition, semiannual workshops were held at UCSD for more comprehensive exchange of results among the researchers funded by DARPA as well as those funded by NSF and DOE in companion evaluations. Presentations on the research results were given at several technical conferences as noted in the section on publications.

Implementation and Evaluation of Defense Applications on MTA

To validate the potential benefits of multithreading, various applications were evaluated on the MTA at SDSC. Included were two large applications, five mini-applications, and seven standard kernels. Another large application was originally planned for study, but was replaced by a mini-application and augmented by kernels to provide broader coverage when it became clear that implementing large applications on the MTA required more effort than anticipated. The applications are described in a later section and in the case studies.

Each collaborating partner was responsible for implementing and evaluating one or more applications. Considerable help in optimizing the applications was provided by Tera.

The generic subtasks originally proposed and carried out by the partners were as follows:

- Become familiar with multithreading and the MTA.
- Develop a multithreaded version of the application that runs on the MTA.
- Identify program bottlenecks and inefficient modules using the Canal and Traceview tools provided by Tera.
- Insert compiler directives or restructure code to improve performance.
- Suggest compiler improvements to eliminate identified problems.
- Devise and implement new algorithms that take better advantage of multithreading.
- Quantify the performance of the application for typical problems on the MTA, with important metrics being processor utilization and scalability.
- Compare performance on the MTA with that achieved on vector multiprocessors, superscalar multiprocessors, and multicomputers.
- Quantify the programming effort and the extent and nature of application changes to achieve superior performance.

Results from all the application case studies were collected by researchers at UCSD and summarized in the performance and usability findings that appear in later sections. Corroborating results have also been obtained by UCSD in its NSF-funded evaluation of the MTA for scientific applications.

Studies of Parallel Profiles and Performance Scaling on MTA

Performance of the applications was examined in considerable detail on the MTA. The intent was to determine whether a multithreaded architecture would support a broader spectrum of parallelism than conventional computers. To quantify this, each of the partners carried out one or more of the following subtasks:

- Profile MTA processor utilization as a function of time using the Traceview tool.
- Determine how many threads are needed for full processor utilization.
- Develop scaling models to project performance to MTA systems larger than the 8-processor system at SDSC.
- Evaluate the efficacy of multithreading to provide high system utilization.

The very late availability of the 8-processor MTA and the presence of broken links in the network greatly reduced the scope of the scaling studies and precluded development of well-substantiated scaling models.

Development and Evaluation of Compilers and Runtime Support for Multithreaded Computers

Compiler technology and runtime support were developed and evaluated by UCSD to improve performance on existing and future multithreaded computers. This work involved the following subtasks:

- Explore new parallelization and code generation techniques that automatically extract parallelism in applications.
- Study the composite parallel profile of real work loads to understand the interaction of concurrent jobs competing for resources of a multithreaded architecture and to support the de-

sign of runtime system strategies that improve processor utilization, job throughput, and load balance.

Additional subtasks to study adaptive execution and compilation strategies were envisioned in the proposal, but were not carried out because access to an MTA with sufficient functionality was delayed.

Tera MTA Characteristics

Multithreaded Processors

The Tera MTA [1] represents a radical departure from traditional vector- or cache-based computers. MTA processors have no data cache or local memory. Instead, they are connected via a network to commodity memory, configured in a shared-memory fashion. Hardware multithreading is used to tolerate high latencies to memory, typically on the order of 150 clock cycles. Even higher latencies can be tolerated by instruction lookahead.

Each processor has up to 128 hardware streams, each of which holds the context for one thread in a program counter and 32 registers. The processor switches from one stream to another every clock period, executing instructions from non-blocked streams in a fair fashion approximating round robin.

A stream can execute an instruction only once every 21 clocks (the length of the instruction pipeline), so a minimum of 21 streams is required to keep a processor fully utilized, even if no instructions reference memory. Typically, 40 or more streams are needed on each processor to hide memory latency and achieve full processor utilization. This requires that the compiler and programmer find substantially more parallelism than on traditional parallel computers. Performance is poor for sections of code with insufficient parallelism and particularly poor for serial sections, which run on a single stream.

If every instruction required data from the previous instruction, then at most 128 clocks of memory latency could be tolerated. However, each stream can issue 8 memory references without waiting for any to return (assuming no dependencies), so instruction lookahead can often sidestep this problem.

The compiler annotates each instruction with a lookahead number: the number of subsequent instructions from the same stream that can be executed before the memory reference of the current instruction must be completed. A lookahead of 0 means that the next instruction needs the current memory reference and cannot be executed until it has completed. If lookahead were always 4, about 30 streams would suffice to saturate the processor. Fortunately, scientific codes exhibit a high level of instruction-level parallelism. The compiler effectively uses software pipelining to schedule memory references ahead of their uses and often achieves the maximum lookahead of 7.

Each clock a processor can issue an instruction containing a memory reference and two other operations. The other operations may be a floating-point add and a floating-point fused multiply-add. Thus the theoretical peak speed of a processor is three floating-point operations per clock (two adds and one multiply). In practice, no more than two floating-point operations per clock (one add and one multiply) has been sustained on realistic computations. The practical peak is very nearly achieved for dot products.

The maximum of a single memory reference per clock sets an upper bound on memory bandwidth of one 8-byte (64-bit) word to each processor every clock. This further limits processor performance for many mathematical kernels. For example, the DAXPY linear algebra kernel is restricted to at most 2/3 of a floating-point operation per clock.

The overhead to acquire a hardware stream is of the order of hundreds of instructions. This affects parallelization strategy and makes performance on small problem sizes disappointing. In general, parallelizing a loop with few instructions and few iterations is counter-productive.

The MTA is able to exploit inner-loop (vector) parallelism, but prefers outer-loop parallelism. The overhead to parallelize an outer loop is paid once, but the overhead to parallelize an inner loop is paid the number of times that the outer loop iterates (unless the loops are fused). Indeed, a good tuning strategy on the MTA is to first find parallelism and then move it to an outer loop. Nevertheless, to get good performance on the MTA, programmers must parallelize at a level that has sufficient parallelism to hide memory latency and amortize overhead costs. This may require parallelization of both inner and outer loops.

The MTA can synchronize at the word level with no additional cost over that of a normal memory reference. This capability enables parallel programming in a dataflow style when dependencies are complicated or not known until runtime.

Network and Memory

The network connecting processors to memory is a partially connected 3-D torus. It is also sparsely populated, with multiple routing nodes per compute processor rather than the reverse configuration increasingly used on distributed-memory systems. Each node has three or four communication ports and a resource port. The resource port may be connected to a compute processor, an I/O processor, or a memory board. Some nodes are not connected to any hardware resource.

The number of nodes is at least $p^{3/2}$, where p is the number of compute processors. This means that the bisection bandwidth scales linearly with p, while the network latency scales as $p^{1/2}$. The maximum bandwidth from memory to a processor is 8 bytes times the clock speed. Small systems have more than $p^{3/2}$ nodes for reasons unrelated to bandwidth requirements.

The 8-processor system at SDSC has 2x4x8 = 64 nodes (where $64 > 8^{3/2} = 22.6$). Of these nodes,

- 8 are attached to compute processors;
- 8 are attached to I/O processors;
- 16 are attached to memory boards;
- 32 are not attached to any resource.

The associated network has 3.5x4x8 = 112 bidirectional links. Several of these links are broken. For memory-intensive applications, the presence of broken links reduces the effective bandwidth per processor as the number of processors increases.

The total amount of memory is proportional to the number of processors. The system at SDSC has one GB per processor corresponding to two memory boards. Each memory board has 64 banks, and memory references by the processors are randomly scattered among all of the banks of all of the memory boards, except for fetches that access the processor instruction cache via a dedicated data path. As a result, memory is equally accessible to each processor, and memory latency is independent of stride.

The MTA thus abstracts away the concept of data layout. The programmer cannot determine the physical allocation of memory and is relieved from concerns associated with the physical representation of an abstract data structure.

Physical Configuration

Processors and memory are combined into four-board resource modules. One board is the compute processor; another board is the I/O processor; and the remaining two boards are for memory. The 8-processor MTA at SDSC thus has 8 resource modules.

The resource modules connect to the backplane, which contains the interconnection network. This network is made up of additional boards, one per resource module, or 8 boards for the machine at SDSC.

Each board contains custom GaAs chips. Overall there are more than 20 different types of chips. Most of the chips on the memory boards, however, are commodity silicon SDRAM.

The MTA is quite compact. Hence it has a high power density and is water cooled.

Software

The Tera operating system is a parallel version of Unix, based on Berkeley sources and modified to use a concurrent microkernel developed by Tera. A two-tier scheduler is incorporated in the microkernel to support execution of multiple tasks, both large and small, without manual intervention. Large tasks (running on more than a single processor) are scheduled via a bin-packing scheme, while small tasks are scheduled using a traditional Unix approach.

Unix became available on the MTA in the latter half of the project. Before that a relatively simple, single-user operating system, called Carlos, was provided.

Fortran, C, and C++ compilers are available. They have separate front-ends, but a common back-end (or optimizer). All three languages may call each other freely.

The sophisticated compilers automatically parallelize serial code by decomposing it into threads. Generally it is necessary for the programmer to augment automatic parallelization by inserting pragmas or explicit parallel constructs into the code. The automatic parallelism uncovered by the compilers and the explicit parallelism described by the programmer can be freely mixed and matched. If desired, the programmer can also take control of thread management to gain finer control of the parallel profile of the application.

Tera provides two powerful tools to help a programmer tune a code for better performance. A tool called Canal provides an annotated version of the source code that shows what the compiler parallelizd, what it did not, and why. A second tool called Traceview does a post-mortem analysis of an execution trace and makes a time-dependent plot of processor utilization and performance.

The combination of multithreaded processors and a uniform shared memory affords a very attractive parallel programming model on the MTA. Multithreading is the only type of parallelization that need be considered, regardless of the number of processors. It can successfully tolerate the latency of memory access, obviating the need for data caches. This relieves the programmer of designing methods for enhancing the locality of memory references.

Evolution of System at SDSC

When the project began in July 1997, a 4-processor MTA was expected to be operational at SDSC within three months, and an upgrade to 8 processors was expected no more than six months later. As it turned out, installation of the 4- and 8-processor systems was delayed by about 15 months because of various manufacturing problems. This necessitated extending the project from 18 months to 24 months to allow 8-processor results to be obtained. Even still, installation of the full 8-processor system was not completed until the last month of the project. This substantial delay and the lack of robust software seriously limited the scope of the evaluation.

To allow the evaluation to begin, Tera delivered smaller one- and two-processor systems during the first year of the project. These had various hardware and software deficiencies, as noted shortly. Some significant milestones in the evolution of the MTA hardware and software at SDSC are indicated in Table 1.

Table 1. Evolution of Tera MTA at SDSC

Delivery date	Processors	Clock (MHz)	Operating system
12/97	1	145	Carlos
1/98	1	260	Carlos
4/98	2	255	Carlos
9/98	2	255	Unix
12/98	4	260	Unix
5/99	8	260	Unix

The clock speed improved substantially from 145 MHz in the initial one-processor system to 260 MHz in the final 8-processor system. This improvement, though welcome, still fell more than 20% short of the design speed of 333 MHz.

Other important hardware characteristics improved with time as manufacturing problems were overcome. In particular, the number of working streams per processor increased from less than 100 in the one- and two-processor systems to more than 100 in the later 4- and 8-processor systems. In addition, scalability was much better on the larger systems, since the presence of broken links in the network was less deleterious.

Significant improvements in the software environment also took place during the project. Carlos, the initial operating system, was superseded by Unix, and the compiler became more powerful as the system matured. Many bugs that plagued early versions of the operating system and compiler were eliminated, resulting in a more robust and productive software environment near the end of the project.

Reference Computer Characteristics

To place in perspective the performance of applications on the Tera MTA, each application was also run on one or more reference computers of more traditional design. These reference computers encompass the major architectural classes. Characteristics of their processors and those of the MTA are listed in Table 2.

Several of the parallel reference computers had more processors than listed in the table. The tabulated numbers are just the maxima used in the comparative calculations. The MTA values correspond to those at the end of the project, with the peak speed based on the practical limit of two floating-point operations per clock.

Table 2. Processor characteristics of computers used in evaluation

Computer class / Model	Processor	Site	Max processors used	Clock speed (MHz)	Peak flops/ clock	Peak processor speed (Mflops)
Shared-memory multiprocessors						
Multithreaded						
Tera MTA	Custom GaAs	SDSC	8	260	2	520
Parallel vector						
Cray T90	Custom ECL	SDSC	8	440	4	1,760
Parallel superscalar						
NeTpower Sparta	Intel Pentium Pro	Caltech	4	200	1	200
Distributed-memory computers						
Uniprocessor nodes						
Cray T3E-600	Digital Alpha 21164	SDSC	64	300	2	600
ccNUMA						
HP X2000	HP PA-8000	Caltech	64	180	4	720
HP V2250	HP PA-8200	Caltech	32	240	4	960
SGI Origin2000	MIPS R10000	Boeing	16	250	2	500
Uniprocessors						
Digital	Digital Alpha 21164A	Caltech	1	500	2	1,000
Sun	Sun UltraSPARC II	SDSC	1	200	2	400

Applications

To validate the potential benefits of multithreading, the original proposal called for at least three large applications and four mini-applications to be evaluated on the MTA. One of the large applications was replaced by a mini-application, and only portions of the other two large applications were considered when it became clear that implementing large applications on the MTA required more effort than anticipated. To assure adequate algorithmic coverage, several standard kernels were added and studied in conjunction with the companion NSF-funded evaluation.

The final applications evaluated, along with the responsible partners, are listed in Table 3. Included are seven standard kernels, five mini-applications, and two large applications. Brief descriptions of each follow here, while more elaborate discussions are included in the case studies.

Table 3. Applications evaluated

Application class / Lead partner	Code	Description	Language	Lines of code
Linear algebra kernels				
SDSC	DGEMM	Matrix-matrix multiply	Fortran	100
	Linpack	Dense linear equation solver	Fortran	700 - 2,000
NPB kernels				
SDSC	CG	Conjugate gradient	Fortran	1,110
	EP	Embarrassingly parallel Monte Carlo	Fortran	250
	FT	Fast Fourier transform	Fortran	1,125
	IS	Integer sort	C	750
	MG	Multigrid	Fortran	1,450
Mini-applications				
SDSC	TPHOT	Monte Carlo photon transport	Fortran	5,000
Caltech	Threat Analysis		C	1,400
	Terrain Masking		C	1,000
	Explode	Thermal explosion model	C	1,000
Sanders	RASSP SAR	High-resolution synthetic aperture radar	C	2,000
Large applications				
Tera	TRANAIR	Computational fluid dynamics	Fortran & C	200,000
Boeing	FMM Prototype	Electromagnetics	Fortran	29,000

One of the kernels added and investigated at SDSC is the Linpack benchmark [2, 3]. It provides an important test for any new computer, since its performance results are widely reported. The kernel consists of a dense linear equation solver common to many scientific computations. The most computationally intensive part of the kernel is the matrix-matrix multiply routine DGEMM [4], which is included here as a separate kernel. Processors frequently sustain a large fraction of their peak performance on DGEMM and Linpack, so these kernels are particularly useful for determining the practical maximum performance of a processor.

The NAS Parallel Benchmarks (NPBs) [5] are also widely studied and reported. They consist of five kernels representing common numerical algorithms and three mini-applications in various sizes. The latest versions of the benchmarks are NPB 2.3 with MPI-based (parallel) source code and NPB 2.3-serial with single-processor (serial) source code. The investigation here, led by SDSC researchers with support from Tera staff, considered only the five kernels and the 2.3-serial source code, which is suitable for machines such as the T90 and MTA that rely on the compiler to do parallelization.

TPHOT is a time-dependent Monte Carlo code that simulates photon transport in a plasma [6]. Various versions of the code have been used for several years to examine and benchmark parallel Monte Carlo particle transport on a variety of computers. TPHOT was added as a mini-application during the course of the project and investigated at SDSC.

Caltech researchers investigated two benchmarks from the C3I Parallel Benchmark Suite [7], which was developed for Rome Air Force Laboratory by Honeywell. The benchmarks considered – Threat Analysis and Terrain Masking – are computationally intensive, memory intensive, and compact. They involve non-trivial data and control structures, and they have the potential for large-scale parallelization. Investigation of a third benchmark was planned, but was deferred when it was decided that the other two were representative.

Caltech researchers also planned to implement and evaluate a large application called SF Express, which simulates synthetic forces of many thousands of entities. Implementation efforts during the first half of the project indicated that the MTA software environment was not mature enough to handle the application's hundreds of files and 500,000 lines of code.

Accordingly, effort at Caltech for the remainder of the project was redirected to a thermal explosion benchmark. This mini-application, called Explode, simulates an explosive wave that initiates chemical reactions in a reactive material [8]. The code employs a fixed 2-D mesh, but has a time-varying compatational workload at each mesh point. Explode thus provides a good test for a parallel computer's ability to do dynamic load balancing.

The final mini-application investigated was the RASSP Synthetic Aperture Radar (SAR) benchmark [9]. Developed at the MIT Lincoln Laboratory, this benchmark was studied by researchers from Sander/Lockheed Martin and Tera as representative of an application that would need to run in real time. The deadline requirements imposed by real-time processing introduce issues not addressed by the other applications in the evaluation.

Tera, with Boeing support, implemented roughly half of TRANAIR [10, 11]. This large, computational fluid dynamics application with about 400,000 lines of code models transonic flow around complex configurations and is used extensively at Boeing for aircraft design. TRANAIR is a general geometry code that solves the full potential equation with a directly coupled integral boundary layer. An adaptive algorithm with a GMRES solver is used. TRANAIR is representative of many defense legacy codes in its complexity and its lack of portability.

Boeing also studied a large, 3-D electromagnetics application, called PARADYM [12, 13]. Its method of moments solver, which uses the GMRES iterative method and the multi-level Fast Multipole Method (FMM), is representative of the best numerical algorithms used at Boeing for computationally intensive applications. The original plan was to implement and tune the full application, which has about 93,000 lines of code. Although a multithreaded implementation was eventually achieved, it worked correctly only on a single processor because of problems with dynamic memory allocation. Thus performance evaluation was focused on the core FMM computation within PARADYM, which did work correctly on multiple processors, except for the largest test case. At 29,000 lines of code, the FMM prototype code is still a large and demanding application.

MTA Performance

Performance results for all of the applications are presented in considerable detail in the case studies later in this report, while early results are in Refs [14] to [19]. A concise summary of the latest results is given in Table 4.

For one or two test problems involving each application, speeds on the MTA are compared with those on a reference computer, both at one processor on each machine and at the number of processors for which the speed has maximized. In most cases, speedups on 4 and 8 processors are also compared. All of the applications have undergone extensive tuning to multithread on the MTA. Where parallel results are available for a reference computer, some tuning on that computer has also been done, but generally not as much as for the MTA.

The relative speed is defined as the ratio of the reciprocal run times on the two machines being compared. The run time measured was the wall-clock (or elapsed) time on a dedicated group of processors, with the exception of some single-processor T90 runs. For those runs, which were not in dedicated mode, the cpu time was measured and closely approximates the wall-clock time in dedicated mode.

Table 4. Performance results for tuned applications on MTA and a reference computer

Application class / Code	Test problem	Reference computer	1p speed: MTA/ref[1]	MTA eff * processors[2]	Ref eff * processors[3]	Max speed: MTA/ref[4]
Linear algebra kernels						
DGEMM	N=2,000	T90	0.31	0.98 * 8		
Linpack	N=1,000	T90	0.16	0.48 * 4		
				0.30 * 8		
	N=10,000	T90	0.26	0.83 * 4		
				0.65 * 8		
NAS parallel kernels						
CG	Class A	T90	0.92	0.94 * 4	0.80 * 4	
				0.72 * 8	0.67 * 8	0.98
	Class B	T90	0.86	0.86 * 8	0.74 * 8	1.00
EP	Class A	T90	1.11	1.00 * 4	1.00 * 4	
				1.01 * 8	1.00 * 8	1.12
	Class B			1.00 * 8		
FT	Class A	T90	0.27	1.01 * 4	0.84 * 4	
				0.95 * 8	0.68 * 8	0.38
	Class B			0.94 * 8		
IS	Class A	T90	1.51	0.94 * 4	0.55 * 4	
				0.78 * 8	0.27 * 8	4.26
	Class B			0.81 * 8		
MG	Class A	T90	0.34	0.95 * 4	0.96 * 4	
				0.81 * 8	0.86 * 8	0.33
	Class B			0.81 * 8		
Mini-applications						
TPHOT	24M photons	T3E	4.19	1.01 * 4	0.99 * 4	
				0.99 * 8	0.99 * 8	
					0.98 *64	0.68
Threat Analysis	5 scenarios	X2000	3.36	0.79 * 4	0.99 * 4	
				0.50 * 8	0.99 * 8	
					0.78 *64	0.27
Terrain Masking	5 scenarios	X2000	5.43	0.97 * 4	0.85 * 4	
				0.71 * 8	0.68 * 8	
					0.50 *12	5.4
Explode	301x301 mesh	V2250	2.50	0.97 * 4	0.61 * 4	
				0.92 * 8	0.43 * 8	
					0.38 *16	3.00
RASSP SAR	One frame of 512 pulses	US II	10.9	0.82 * 4		
				0.69 * 8		
Large applications						
TRANAIR	B747I-4: 115k rows	T90	0.45	0.71 * 4		
				0.60 * 8		2.15
	B747I-5: 139k rows	T90	0.37	0.81 * 4		
				0.60 * 8		1.78
FMM Prototype	8x8 plate: 16k sources	O2000	0.63	0.89 * 4	0.97 * 4	
				0.66 * 8	0.91 * 8	
					0.62 *16	0.34
	16x16 plate: 64k sources	O2000	0.83	0.84 * 4	0.97 * 4	
				0.69 * 8	0.91 * 8	
					0.80 *16	0.36

1. Ratio of single-processor speed on MTA to that on reference computer
2. MTA speedup = efficiency * number of processors on MTA
3. Reference speedup = efficiency * number of processors on reference computer
4. Ratio of maximum speed on MTA to that on reference computer

The definitions of speedup and parallel efficiency are standard: speedup is just the ratio of the reciprocal run time on a specified number of processors to that on one processor, while the parallel efficiency is the ratio of the speedup to the number of processors. For those few instances in which the reference computer had a single-processor speed for parallel code measurably lower than for serial code, the speedup is relative to the serial code. Thus the speedups may appear lower in Table 4 than in some of the case study tables.

Single-Processor Performance

Single-processor speeds on the MTA and T90 are compared in Table 4 for seven standard kernels and one large application. Six of the kernels are written in Fortran; one kernel is written in C; and the large application is mostly Fortran, with a small amount of C.

On four kernels – DGEMM, Linpack, FT, and MG – and the large TRANAIR application a single T90 processor is appreciably faster than one MTA processor. These codes vectorize very well on the T90, which allows it to take advantage of its much higher peak floating-point performance and larger number of memory accesses per clock compared to the MTA. The measured speed advantage of the T90 varies from a factor of 6.2 for the Linpack N=1,000 problem down to a factor of 2.2 for one of the TRANAIR test problems.

For two kernels – CG and EP – the single-processor speeds on each computer are within 15% of each other. For one kernel – IS – the MTA is appreciably faster than the T90, by a factor of 1.5. This is the only kernel written in C, though that is probably coincidental.

Five of the kernels are floating-point intensive. For these it is useful to compare the measured T90 and MTA speeds with the peak processor speeds. Such a comparison is shown in Table 5.

Table 5. Single-processor speeds for floating-point intensive kernels on T90 and MTA

Kernel	Problem size	Measured T90 speed (Mflops)	Measured MTA speed (Mflops)	Measured/ peak T90 speed	Measured/ peak MTA speed
DGEMM	N=2,000	1,580	495	0.90	0.95
Linpack	N=1,000	1,441	232	0.84	0.45
	N=10,000	1,556	411	0.88	0.79
CG	Class A	178	164	0.10	0.32
	Class B	202	174	0.11	0.33
FT	Class A	696	187	0.40	0.36
MG	Class A	563	194	0.32	0.37

The peak processor speeds are 1,760 Mflops (based on four flops per clock) on the T90, as compared to 520 Mflops (or two flops per clock) on the MTA. The T90 also has higher memory bandwidth, since it can issue four loads and two stores each clock (with its two pipes) versus only one memory reference per clock on the MTA.

For three of the kernels – DGEMM, FT, and MG – the two machines achieve comparable fractions of their peak speeds. For Linpack the T90 gets a substantially higher fraction of its peak speed than does the MTA (especially for small problems), whereas for CG the reverse is true (because of poor vectorization on the T90). Overall, the MTA shows less variability in the fraction of peak speed achieved, which suggests that multithreading may be more broadly applicable than vectorization.

On the other hand, the markedly lower performance on the MTA of Linpack compared to its dominant DGEMM kernel points up a significant problem. MTA performance is evidently quite sensitive to small regions of code where less than optimal parallelization can be achieved. Such regions

occur in the final stages of dense matrix factorization in Linpack and in various routines other than the dominant DGEMM kernel.

Single-processor speeds on the MTA and several computers with workstation processors are compared in Table 4 for five mini-applications and one large application. Four of the mini-applications are written in C, while the remaining mini-application and large application are in Fortran.

For all of the mini-applications – THPOT, Threat Analysis, Terrain Masking, Explode, and SAR – the MTA is faster on a single-processor basis. The largest MTA speed advantage is more than a factor of 10 relative to a 200-MHz UltraSPARC II for the SAR benchmark. The smallest advantage is a factor of 2.5 relative to a 240-MHz PA-8200 in the V2250 for the Explode benchmark, which is written in C.

It is noteworthy that the MTA substantially outperforms cache-based processors with comparable clocks for all of the mini-applications. This indicates the effectiveness of multithreading as compared to cache utilization for modest size applications.

By contrast, for the FMM large application a single MTA processor is slower than a 250-MHz MIPS processor in the O2000. The speed advantage of the O2000 is a factor of 1.6 for the smaller problem and a factor of 1.2 for the larger problem. Evidently the FMM Prototype has substantial data locality, which allows very effective use of cache on the O2000.

Multithreading (i.e., parallelization) is required on the MTA to get acceptable performance even on a single processor. Performance of serial sections of code is very slow on the MTA - slower than on a typical workstation – so tuning to minimize such sections is critical.

The importance of tuning is illustrated by the results presented in the case study on the two C3I mini-applications. The untuned versions of the Threat Analysis and Terrain Masking codes were effectively entirely serial and so ran 8.9 and 4.9 times *slower*, respectively, on an MTA processor than on a 180-MHz PA-2000 processor in the X2000. After tuning for parallelization, the same codes were 30 and 26 times *faster* on a single MTA processor than before and 3.4 and 5.4 times *faster* than on an X2000 processor.

Even for these relatively small mini-applications the sophisticated Tera C compiler is unable to achieve any performance improvement through automatic parallelization. These codes contain loops with embedded function calls and pointer references. Manual tuning by restructuring the loops or inserting compiler directives is required to parallelize such codes.

Scalability

The results in Table 4 allow comparison of scalability on 8 processors of the MTA and a reference computer for ten of the applications. No systematic difference is apparent. Scalability on the MTA is much better than on the reference computer for three applications – FT, IS, and Explode – and much worse for two others – FMM Prototype and Threat Analysis. For the other five applications – CG, EP, MG, TPHOT, and Terrain Masking – the differences in scalability between the MTA and the reference computer are modest or negligible.

For one more application – TRANAIR – scalability on the MTA is implicitly better than on the T90, since parallelization of this application is so difficult on the T90 that it has not proved practical. On the other hand, scalability of TRANAIR on the MTA is still relatively poor, and roughly three MTA processors are required to match the speed of a single T90 processor.

Table 4 also contains sufficient information to classify MTA scalability for all 14 applications. Such a classification is shown in Table 6, where the applications are sorted by parallel efficiency on 8 MTA processors. Where an application has results for two test problems, the better efficiency is listed in Table 6.

This classification shows a clear difference in scalability between application classes: scalability of kernels is appreciably better than for the mini- and large applications. Indeed, 6 out of 7 kernels

Table 6. Scalability of each application on 8 MTA processors

Scalability class	Efficiency at 8p	Application name	Application class
Very good	1.00	EP	Kernel
	0.99	TPHOT	Mini-application
	0.98	DGEMM	Kernel
	0.94	FT	Kernel
	0.92	Explode	Mini-application
Good	0.86	CG	Kernel
	0.81	IS	Kernel
	0.81	MG	Kernel
Fair	0.71	Terrain Masking	Mini-application
	0.69	SAR	Mini-application
	0.69	FMM Prototype	Large application
	0.65	Linpack	Kernel
Poor	0.60	TRANAIR	Large application
	0.50	Threat Analysis	Mini-application

have good or very good scalability, while 5 out of 7 of the mini- and large applications have fair or poor scalability.

There are various reasons for the limited scalability on the MTA of the lowest ranked applications in Table 6. In the case of the Threat Analysis, Terrain Masking, and SAR benchmarks the test problems may simply be too small. For Linpack, scalability is limited by decreasing parallelism near the end of the computation. As for TRANAIR and the FMM Prototype, these large applications are inherently more difficult to parallelize. Further tuning of them is needed and should improve their scalability. Simply increasing the problem sizes for them is likely to be of little benefit.

Extrapolation of MTA performance to larger systems was considered for several applications. However, because of frequent changes in the hardware and software environment, sufficient data could not be collected to construct convincing models, with possibly one exception. For Explode and its test problem, the parallel efficiency on the MTA is fit very well by Amdahl's law with a small serial fraction of 0.013. This corresponds to a single overhead due to serial code. For the other applications, more complicated scaling models are required. Besides serial overhead, one would need to consider various combinations of other overheads due to load imbalance, thread creation, memory contention, and network contention.

Overall Performance

Ultimately what is of interest is the overall performance, which can be obtained by combining single-processor performance and scalability data. The last column in Table 4 gives the ratio of the maximum speed on the MTA to that on the reference computer for 11 applications: all ten that were parallelized on the reference computer plus TRANAIR. Since TRANAIR is very difficult to parallelize on the T90, comparing multiple MTA processors to a single T90 processor seems fair. The maximum speed on each computer was usually, but not always at the largest number of processors listed.

For 4 of the 11 applications – IS, Terrain Masking, Explode, and TRANAIR – the MTA is appreciably faster than the reference high-end computer. The speed advantage of the MTA ranges from a factor of 1.8 over the T90 for one of the TRANAIR problems to a factor of 5.4 over the X2000 for Terrain Masking.

The maximum performance on the MTA is at the full 8 processors available, except for Terrain Masking where performance peaks at 7 processors. By contrast, performance on the reference computers peaks before the full configuration is reached: at 4 T90 processors for IS, at 12 X2000

processors for Terrain Masking, and at 16 V2250 processors for Explode. In the case of IS, Explode, and TRANAIR, the scalability advantage of the MTA contributes substantially to the overall performance advantage.

For two other applications – CG and EP – the speed on 8 MTA processors is comparable to the speed on 8 T90 processors. These kernel applications have similar single-processor performance and scalability on each computer.

MTA Usability

An important goal of this project was to evaluate the overall usability of the Tera MTA. Raw performance measures are more meaningful when interpreted in context of the programming effort required to achieve them. The MTA presents a new programming model along with new hardware, software, and programming tools. It is important to evaluate each.

It is convenient to consider in order:

- Programmer effort and productivity,
- Effectiveness and quality of tools,
- Reliability of hardware and software,
- Viability of system for achieving scientific progress.

Programmer Level of Effort

Multithreading as implemented by Tera in the MTA offers a single parallel programming model, which is the same for multiple processors as for a single processor. Programming and tuning for multithreaded performance can be done on a single processor and then simply extended to multiple processors, provided the problem contains sufficient parallelism. This simple parallelization model was very well received by the participants in the evaluation.

By contrast, to get multiprocessor speedup on a traditional vector multiprocessor with shared memory, one must vectorize inner loops to gain single-processor performance and then multitask across processors. Similarly, on a distributed-memory computer built of cache-based processors, one must tune for data locality and exploit instruction-level parallelism to get good single-processor performance, then distribute the data and use something such as MPI to get the communication right for multiple processors. The single parallelization model of the MTA, whether for one or many processors, is a major benefit.

The evaluation here focused on porting and tuning existing codes rather than programming new ones. It was found that starting from a serial formulation of an algorithm or from source code (without MPI) suitable for a shared-memory machine often led to a straightforward port. Tuning such codes sometimes included stripping out data-moving instructions meant to benefit machines with cache or vector units (where data layout matters), because such instructions serve no useful purpose on the MTA. Porting MPI codes to the MTA was avoided, since that was expected to be fairly labor-intensive. Communication calls and work to distribute data would need to be removed while preserving the semantics of the program.

Porting small applications to the MTA was found roughly comparable to porting them to other uniform shared-memory machines and easier than to distributed-memory machines (except for the embarrassingly parallel TPHOT application). The programmer does not have to be concerned about data distribution across hierarchical memory. With no concept of local or remote memory, the MTA relieves the programmer of this burden.

Tuning small applications for the MTA was also found easier than on distributed-memory machines. With no data cache and no concept of data locality on the MTA (since memory addresses are randomized across memory modules), the programmer is relieved of the responsibility of optimizing memory-access patterns for cache.

On the other hand, porting and tuning large applications to run on the MTA were found to be much more difficult. Strict conformance of the compilers to language conventions means that just getting large codes to compile error-free takes a long time. Extensive tuning is then required. Since the MTA requires high levels of parallelism, more sections of serial code need to be parallelized than on other computers, and more parallelism must be found. This, coupled with the greater complexity of the applications, takes a long time.

Effectiveness and Quality of Tools

The MTA comes with sophisticated compilers that attempt to parallelize automatically a program written in the traditional serial languages of Fortran, C, and C++. The compilers are powerful tools, and the underlying technology is extremely impressive. Nevertheless, the C compiler was found largely unable to parallelize code automatically. While the Fortran compiler does much better, one can seldom simply compile serial code and expect good performance "out of the box."

Rather, the programmer must tune the code in a feedback loop using the following four tools:

1) Canal, a tool that produces an annotated listing of the compiled code to show what the compiler parallelized, what it failed to parallelize, and why;
2) A set of directives that allow the programmer to "coax" the compiler to parallelize;
3) Extensions to the languages allowing explicit programming of parallel constructs;
4) Traceview, a tool that allows the user to capture and view time-dependent execution profiles.

These tools were found to be useful and effective. The programmer compiles the code, sees what the compiler has done via Canal, and iterates until directives or code restructuring make the executable look reasonably parallel. Then the programmer runs the code with tracing enabled. If performance is less than anticipated, Traceview will show where (usually a serial section or loop with insufficient parallelism). The programmer then goes back to work with the compiler on the problem area.

The feedback loop was seriously disrupted during the evaluation period by system and compiler irregularities. These made it difficult to separate bugs introduced during tuning from those in the operating system or compiler. Compounding this were frequent software upgrades. These upgrades generally added functionality and fixed some problems, but often introduced other problems. Moreover, researchers had to recompile and rerun performance tests after most upgrades.

Also, execution times were not always reproducible. Because system resources are shared cycle-by-cycle on the MTA, the execution of one program is very sensitive to the activities of another, including those spawned by the operating system. As the load on the machine increased, tuning became more difficult, and dedicated time slots were essential for getting useful performance information.

Additional tools provided by Tera include libraries for common mathematical operations, such as the BLAS. Their usefulness in the evaluation was limited, in part because of their performance varied significantly as the compiler evolved.

Reliability of Hardware and Software

As noted in Table 1, Unix was not available on the MTA until September 1998. Before then, use of the machine was restricted to relatively few users, primarily running small applications.

With the availability of Unix, serious work began on all of the applications discussed here. Even still, various system problems made use of the MTA difficult. Of particular concern were

- operating system instability,
- lack of system swapping,
- compiler bugs and "features,"
- slow I/O.

These problems have been significantly mitigated in recent months, but further improvement is needed.

The system has become much more stable, with the mean time between interrupts growing from one to many hours. Swapping was recently implemented, which should eliminate the memory allocation problems that were common during the evaluation. The compilers continue to evolve, as many bugs are fixed, and a few new ones are introduced. I/O by ftp to and from a workstation remains slow, but a workaround through faster I/O paths is available for large files.

Executables continue to be cross-compiled on a Sun workstation and then moved by ftp to the MTA. This is tolerable with the recent improvements in system stability and I/O performance. Native compilation on the MTA recently became available, but is very slow because it runs primarily in serial mode.

Viability of System for Scientific Progress

The MTA shows much promise. Indeed, it runs some science and engineering codes as fast or faster than any other supercomputer at SDSC, as shown here and in a companion NSF-funded evaluation. However, because of the previously mentioned reliability problems, the MTA has not yet been used for significant computational science. The work so far has focused on software development, evaluation, and computer science studies. A more robust software environment is needed to support production use.

Multithreaded Scheduler and Compiler Studies

Several studies were carried out related to job scheduler and compiler effectiveness on multithreaded computers.

Symbiotic Job Scheduling for High Throughput

Multithreaded computers not only provide an opportunity for increased performance when executing a single application, but they also provide for increased system throughput. The parallel profiles of applications vary substantially with time. Sometimes a parallel application uses hundreds or thousands of threads, while at other times it can only effectively use one or a few threads.

Multithreaded computers with their lightweight contexts and ability to easily associate contexts with different applications provide a realistic opportunity to share a large computing resource more effectively; parallel peaks in one application's profile can fill parallel valleys in another's profile.

The term *symbiosis* has been adopted to refer to the increase in throughput that can occur when two or more jobs simultaneously finish more quickly than when run in succession on a multithreaded computer, even on a single processor. This is possible on the Tera MTA, for example, if streams unused by one job can be used by a coscheduled job.

Improved throughput via symbiosis has been demonstrated on the MTA for various combinations of jobs consisting of the NAS Parallel Benchmark kernels. Details are given in one of the case studies and in Ref. [20]. The principal findings are the following:

- Serial codes are highly symbiotic with each other and with parallel codes on the MTA. Thus, poor serial performance on the MTA need not impact system throughput as long as serial jobs are coscheduled.

- Highly tuned codes are less symbiotic than untuned versions of the same codes on the MTA. Because a parallel section cannot acquire streams released by another parallel section, highly tuned parallel codes with few serial sections can "throttle" each other and lead to reduced throughput or "negative" symbiosis. A more dynamic resource allocation strategy is being designed to address this issue.

Task-Based Parallelism in Irregular Applications

Parallel computers, including the Tera MTA, depend heavily on compiler-generated parallelism. Currently, compilers rely principally on loop-level parallelism (LLP), where a program benefits from executing different iterations of the loop in different threads. However, a large number of applications have little or no LLP available. For such irregular applications another source of parallelism is needed, namely task-level parallelism.

Task-level parallelism arises when a sequence of instructions (a task) is found independent of neighboring instructions and so can be executed in parallel. The task could be, for example, a procedure call. LLP is a special case of task-level parallelism if the task is a single loop iteration.

One project study has investigated leaf procedures (procedures that make no other procedure calls), non-leaf procedures, and entire loops as possible task boundaries for compilation. The assumption is made that some speculation can be tolerated; that is, a mechanism exists for resolving incorrect speculation when the task is not, after all, independent. This allows speculative parallelization of code when there is a high probability, but no guarantee of independence.

Because the biggest barrier to detecting independence in irregular codes is memory disambiguation, a profile-based tool [21] has been written to identify memory-independent tasks, specifically tasks that have no memory-dependence conflicts with the preceding code. Such tasks can be executed earlier than they appear in the source code, in parallel with some of the preceding code. The tool provides information that could be used in several architectures, including multithreaded ones, to identify possible sources of task-level parallelism. For varying assumptions about the underlying architecture, from 7 to 22 percent of the instructions was found to be within tasks that are memory independent [22]. This is on a set of irregular applications, for which traditional methods of parallelization are very ineffective.

Thread-Level versus Instruction-Level Parallelism

All modern pipelined processors require the compiler to find independent instructions that can be in the pipeline simultaneously – so-called instruction-level parallelism (ILP). In multithreaded architectures, such as the Tera MTA and Simultaneous Multithreading (SMT) architecture, there is a competing requirement of finding thread-level parallelism (TLP) – independent instructions to make up the separate threads of control. The compiler must decide how much of the available parallelism to "spend" on each of these two requirements.

A detailed examination [23] has been made of the tradeoff between using parallelism for ILP and TLP on an SMT architecture, where the problem is particularly difficult (even more than on the MTA) due to the confounding issue that threads share registers and cache. Results obtained suggest that there is no easy answer, but that the optimal choice needs to be based on multiple criteria, including the communication-to-computation ratio, cache and TLB locality, the demand for register names, and the demand for processor resources.

MTA Compiler Evaluation

Throughout the project the quality of object code produced by the optimizer of the Tera compilers was evaluated. The optimizer faces a formidable challenge, since it must generate code that exploits parallelism at two levels: ILP and TLP. At the thread level, it must find enough parallelism to use a sufficient number of streams to hide memory latency. Using both Canal reports and the generated assembly code, the optimizer's results were examined for kernels from linear algebra, fluid dynamics, fast Fourier transforms, and seismic migration applications.

In general, the results are state of the art and most impressive. The optimizer often generates optimal code, meaning that it issues one memory reference per instruction (the maximum possible) and effectively uses software pipelining to achieve lookahead 7 (again the maximum possible). The generated code also uses an effective and low-overhead variant of dynamic scheduling to distribute iterations of a parallel loop to processors. To increase parallelism the optimizer fuses multiple levels of nested loops. It can fuse most rectangular and triangular loop nests. In cases where fusion is

not possible, it may be necessary to generate multiple compiled variants and have a mechanism that chooses the appropriate variant at run time.

Acknowledgments

Many collaborators besides those listed on the front of this report contributed to the work reported here. Included are all the authors of the publications listed in the following section, as well as the authors noted in the case studies. Their assistance is gratefully acknowledged.

This work was supported by DARPA contract DABT63-97-C-0028. Some work at UCSD was jointly supported by NSF grant ASC-9613855.

References

Background

1. www.tera.com/www/library/index.html.

2. J. Dongarra, J. Bunch, C Moler, and G. Stewart, LINPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia PA (1979).

3. www.netlib.org/benchmark/performance.ps.

4. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, LAPACK Users' Guide, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia PA (1995).

5. www.nas.nasa.gov/Software/NPB.

6. W. R. Martin, T. C. Wan, T. S. Abdel-Rahman, and T. N. Mudge, "Monte Carlo Photon Transport on Shared Memory and Distributed Memory Parallel Processors," *International Journal of Supercomputer Applications*, 1(3), 57 (1987).

7. R.C. Metzger, B. VanVoorst, L.S. Pires, R. Jha, W. Au, M. Amin, D.A. Catanon, and V. Kumar, "The C3I Parallel Benchmark Suite – Introduction and Preliminary Results," *Proceedings of SC96*, Pittsburgh PA (November 1996).

8. M. Short and J.J. Quirk, "On the Nonlinear Stability and Detonability Limit of a Detonation Wave for a Model Three-Step Chain-Branching Reaction," *Journal of Fluid Mechanics*, 339, 89-112 (1997).

9. B.W. Zuerndorfer et al., "RASSP Benchmark-1 Technical Description," MIT Lincoln Laboratory Project Report RASSP-1 (December 1994); also llex.ll.mit.edu/llrassp/documents.html.

10. D.P. Young, R.G. Melvin, M.B. Bieterman, F.T. Johnson, S.S. Samant, and J.E. Bussoletti, "A locally refined rectangular grid finite element method: Application to computational fluid dynamics and computational physics," *J. Comp. Phys.*, 92(1), 1-66 (1991).

11. F.T. Johnson, S.S. Samant, M.B. Bieterman, R.G. Melvin, D.P. Young, J.E. Bussoletti, and C.H. Hilmes, "TranAir: A Full-Potential, Solution-Adaptive, Rectangular Grid Code for Predicting Subsonic, Transonic, and Supersonic Flows About Arbitrary Configurations," NASA Contractor Report 4348 (December 1992).

12. R. Coifman, V. Rokhlin and S. Wandzura, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription", *IEEE Antennas and Propagation Magazine*, 35(3), 7-12 (June 1993).

13. B. Dembart and E. L. Yip, "A 3-D Fast Multipole Method for Electromagnetics with Multiple Levels", ISSTECH-97-004, The Boeing Company (December 1994).

Publications

Publications resulting from the project are listed here. Several can be obtained at www.sdsc.edu/~allans/papers.html.

14. J. Boisseau, L. Carter, K. Gatlin, A. Majumdar, and A. Snavely, "NAS Benchmarks on the Tera MTA," *Proceedings of Workshop on Multi-Threaded Execution, Architecture, and Compilers*, Las Vegas NV (January 1998).

15. J. Boisseau, L. Carter, A. Snavely, D. Callahan, J. Feo, S. Kahan, and Z. Wu, "Cray T90 vs. Tera MTA: The Old Champ Faces a New Challenger," *Proceedings of Cray User Group Conference*, Stuttgart, Germany (June 1998).

16. A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K.S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-Processor Performance on the Tera MTA," *Proceedings of SC98*, Orlando FL (November 1998).

17. S. Brunett, J. Thornley, and M. Ellenbecker, "An Initial Evaluation of the Tera Multithreaded Architecture and Programming System Using the C3I Parallel Benchmark Suite," *Proceedings of SC98*, Orlando FL (November 1998).

18. L. Carter, J. Feo, and A. Snavely, "Performance and Programming Experience on the Tera MTA," *Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio TX (March 1999).

19. L.Y. Zaslavsky, S.H. Kahan, B.H. Elton, K.J. Maschhoff, and L.G. Stern, "A Scalable Approach for Solving Irregular Sparse Linear Systems on the Tera MTA Multithreaded Parallel Shared-Memory Computer," *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio TX (March 1999).

20. A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "Explorations in Symbiosis on Two Multithreaded Architectures," *Proceedings of Workshop on Multi-Threaded Execution, Architecture, and Compilers*, Orlando FL (January 1999).

21. B. Kreaseck, D. Tullsen, and B. Calder, "PTIC: A Tool for Finding Parallel Tasks in Irregular Codes," UCSD Department of Computer Science and Engineering Technical Report CS99-626 (July 1999).

22. B. Kreaseck, D. Tullsen, and B. Calder, "Limits of Task-based Parallelism in Irregular Applications", UCSD Department of Computer Science and Engineering Technical Report CS99-627 (July 1999).

23. N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "ILP versus TLP on SMT", *Proceedings of SC99*, Portland OR (November 1999: to appear).

CASE STUDIES

Linear Algebra Kernels

Robert Leary
San Diego Supercomputer Center

Linear algebra is central to many scientific computations. Accordingly, the performance of various dense linear algebra kernels was measured, optimized, and analyzed on up to eight processors of the Tera MTA. Performance of these kernels was also compared with that on a single processor of the Cray T90.

DGEMM – Matrix-Matrix Multiply

The limiting floating-point performance on the MTA for any linear algebra kernel in which additions and multiplications are asymptotically balanced is 2 flops/clock (or 520 Mflops per processor for the current 260-MHz clock rate). Such balanced kernels include the vector dot product, the DAXPY vector operation, matrix-vector multiplication, and conventional matrix-matrix multiplication. Higher rates can only be achieved with parallel use of the fused multiply-add and the separate hardware add operations, thus implying an unbalanced kernel with more additions than multiplications.

On many computers the highest sustained floating point performance is achieved with a conventional matrix-matrix multiply, usually coded in assembly language and highly tuned to the specific hardware. Interestingly, on a single MTA processor a simple Fortran matrix-matrix kernel written in the traditional row-by-column form using a dot-product inner loop achieves 490 Mflops (94% of theoretical peak) for square matrices of order N≥1,000. The $R_{1/2}$ value at which half the theoretical peak is obtained occurs near N=30. The Canal analysis tool indicates that the kernel loop is indeed achieving 16 floating-point operations per eight memory references, consistent with 2 flops/cycle.

Similarly, the best observed matrix-vector multiplication coding is the most primitive Fortran code with a dot-product inner loop. Here Canal indicates 16 flops per 9 memory references, or a limiting rate of 8/9 (89%) that of matrix-matrix multiplication. The observed rate on one MTA processor for a square matrix with N=1,000 is 435 Mflops, or 89% of the observed matrix-matrix multiplication rate, in excellent agreement with the Canal analysis.

All optimization attempts using the conventional loop transformations (e.g., unrolling, blocking, or use of vector temporary arrays) that are applicable to vector and cache-based scalar processors proved counter-productive. This indicates that the compiler is producing essentially optimal code from the simplest Fortran kernels.

The first two rows of Table A-1 compare single-processor performance on the MTA and T90 for a matrix-matrix multiply of order N=2,000 using DGEMM, a routine from LAPACK [A-1]. With appropriate coding, DGEMM achieves close to the practical peak on each machine. The DAXPY formulation is used on the T90, while the dot product formulation is used on the MTA. The T90 requires inner-loop parallelism, while the MTA prefers (but does not require) outer-loop parallelism. This shows that codes that are ported from a vector machine to the MTA may require restructuring to achieve optimal performance. Table A-1 further illustrates that the MTA will not outperform the T90 on highly vectorizable codes.

To execute on multiple processors, MTA code needs to be compiled in *crew* mode as opposed to *fray* mode, which is restricted to a single processor. Initial tests showed that crew code for DGEMM ran significantly slower on a single processor than did fray code. This is apparent from the third row of Table A-1, where the crew interleaved speed is 440 Mflops, as compared to 490 Mflops for the fray speed. This discrepancy was eventually traced to differences in the algorithm used to schedule threads.

Table A-1. Performance results for DGEMM with N=2,000

Computer & scheduler option	Processors used / total	Speed (Mflops)	Speedup	Efficiency
T90	1/	1,580		
MTA fray	1/ 4	490		
MTA crew interleaved	1/ 4	440	1.00	1.00
	2/ 4	875	1.99	0.99
	4/ 4	1,744	3.96	0.99
MTA crew dynamic	1/ 4	487	1.00	1.00
	2/ 4	953	1.96	0.98
	4/ 4	1,902	3.91	0.98
	1/ 8	495	1.00	1.00
	2/ 8	980	1.98	0.99
	4/ 8	1,959	3.86	0.99
	8/ 8	3,870	7.82	0.98

At the time of the initial DGEMM runs, the default was dynamic scheduling for fray mode and interleaved scheduling for crew mode. This turned out to be a poor choice. When a compiler directive was inserted to force dynamic scheduling, crew mode performance improved to 487 Mflops. This is essentially the same as in fray mode and demonstrates very little overhead for running in crew mode.

Interleaved scheduling gives a fixed number of loop iterations to each thread. Dynamic scheduling causes an executing thread to ask for more work when it finishes its current allotment. Dynamic scheduling costs a bit more in terms of software overhead, but results in better load balancing if the work of distinct loop iterations is variable. As a result of the experience with DGEMM (and other kernels), dynamic scheduling has now been made the default in the MTA compiler.

For both scheduling algorithms the scalability of DGEMM on the MTA is excellent. This is apparent from the multiprocessor results shown in Table A-1. The first sets of data are for the 4-processor MTA system, while the last set of data is for the 8-processor system that was installed late in the project. For the same number of processors, the results are 2 to 3% better on the 8-processor system than on the 4-processor system.

Linpack

Linpack [A-2, A-3] solves dense systems of linear equations using routines from the BLAS. For the N=100 test, no tuning is allowed, and the level 1 BLAS are used. For the larger, N≥1,000 tests, tuned versions of the level 2 and level 3 BLAS are used, with the majority of the execution time spent in DGEMM. Table A-2 shows comparative performance for several Linpack tests on the MTA and T90.

For the N=100 test (without tuning), the DAXPY version of matrix-matrix multiply is used. This is not well suited to the MTA. This and the lack of sufficient parallelism in this small problem lead to disappointing results. Both the MTA and T90 run well below peak, but the MTA performance is especially low.

For the larger Linpack problems, performance on both the MTA and T90 is much better. However, as suggested by the results in Table A-2, the asymptotic speed is approached much more slowly with increasing N on the MTA than on the T90.

For the larger problems, the dominant DGEMM matrix-matrix multiply has dimensions M by NB and NB by M, where NB is a fixed but tunable blocking factor, and M decreases from a starting size of N in decrements of NB as the computation progresses. With some experimentation, the best blocking size on the MTA was found to be about NB=100 for large matrices of order N=7,000, resulting in a speed of 425 Mflops at this size. (For reference, the N=100 matrix multiply DGEMM speed is 460 Mflops.) For smaller matrices, such as in the N=1,000 Linpack benchmark, the optimal blocking size was considerably smaller at NB=40, and the best speed observed was 288 Mflops.

Table A-2. Performance results for Linpack

Test problem	Computer & scheduler option	Processors used / total	Speed (Mflops)	Speedup	Efficiency
N=100 untuned	T90	1/	464		
	MTA fray	1/ 4	29		
N=1,000	T90	1/	1,441		
	MTA fray	1/ 4	288		
	MTA crew interleaved	1/ 4	241	1.00	1.00
		2/ 4	334	1.39	0.69
		4/ 4	421	1.75	0.44
	MTA crew dynamic	1/ 4	278	1.00	1.00
		2/ 4	418	1.50	0.75
		4/ 4	546	1.96	0.49
		1/ 8	232	1.00	1.00
		2/ 8	332	1.43	0.72
		4/ 8	445	1.92	0.48
		8/ 8	551	2.38	0.30
N=4,000	T90	1/	1,550		
N=7,000	MTA fray	1/ 4	425		
	MTA crew interleaved	1/ 4	374	1.00	1.00
		2/ 4	728	1.95	0.97
	MTA crew dynamic	1/ 4	412	1.00	1.00
		2/ 4	761	1.85	0.92
		4/ 4	1,290	3.13	0.78
N=10,000	T90	1/	1,556		
	MTA crew interleaved	4/ 4	1,353		
	MTA crew dynamic	1/ 8	411	1.00	1.00
		2/ 8	777	1.89	0.95
		4/ 8	1,359	3.31	0.83
		8/ 8	2,124	5.17	0.65

MTA performance is evidently quite sensitive to small regions of code where less than optimal parallelization can be achieved. Such regions occur in the final stages of dense matrix factorization and in the various Level 1 and Level 2 BLAS routines that are employed in addition to the dominant Level 3 DGEMM kernel. The proportion of time spent in such regions decreases with N, thus accounting for the increase in performance with N. However, the difference between the N=1,000 and N=10,000 performance is much larger than that on the T90, where the difference is less than 10%. This indicates a heightened sensitivity of the MTA to such effects.

The change to dynamic scheduling also improved the Linpack results, since DGEMM is the dominant computational kernel. For example, for the N=1,000 case, the one-processor crew results improved by 15% from 241 to 278 Mflops on the 4-processor MTA. Some of the improvement also appears to be related to improved performance of routines other than DGEMM, including the Tera BLAS Level 1 and 2 library routines.

The move from the 4-processor MTA to the 8-processor MTA had little effect on the Linpack speed per processor for the N=10,000 case, but actually decreased the speed per processor by 20 to 25% for the N=1,000 case. The reason for this anomaly is unknown, but is presumably due to some compiler change on the 8-processor system that affects part of the test code other than DGEMM.

References

A-1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, LAPACK Users' Guide, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia PA (1995).

A-2. J. Dongarra, J. Bunch, C Moler, and G. Stewart, LINPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia PA (1979).

A-3. www.netlib.org/benchmark/performance.ps.

NAS Parallel Benchmark Kernels

Allan Snavely, Jay Boisseau, Larry Carter, Kang Su Gatlin, Amit Majumdar, Laura Nett, and Nick Mitchell
San Diego Supercomputer Center &
Computer Science and Engineering Department
University of California, San Diego

John Feo and Brian Koblenz, Tera Computer

Performance of the Tera MTA and Cray T90 has been compared for the NAS Parallel Benchmark kernels. Such a comparison is useful because the T90 shares important attributes with the MTA: an absence of memory hierarchy, a sophisticated automatic parallelizing compiler, and a processor architecture custom designed for high performance computing. These attributes offer the possibility of a simpler, shared-memory programming model than the message-passing model common on distributed-memory computers. Thus, the evaluation also included an assessment of the effort required to program and tune programs for the MTA.

Processor Characteristics

To facilitate comparison of the MTA and T90, some pertinent processor characteristics are given in Table B-1.

Table B-1. Processor characteristics of the T90 and MTA

Characteristic	Cray T90	Tera MTA
Clock speed (MHz)	440	260
Practical peak flops/clock	4	2
Practical peak Mflops	1,760	520
Memory accesses per clock	2 loads + 1 store	1
Special registers	8 128-element vector registers	128 32-register streams

NAS Parallel Benchmarks (NPBs)

The NAS Parallel Benchmarks [B-1] consist of five kernels and three mini-applications in several problem sizes. The latest versions of the benchmarks are NPB 2.3 with MPI-based (parallel) source code and NPB 2.3-serial with single-processor (serial) source code. The investigation here considered only the

- five kernels,
- 2.3-serial source code, which is suitable for machines such as the T90 and MTA that rely on the compiler to do parallelization,
- Class A and B sizes (except for one Class W case).

Various levels of tuning were applied to each kernel on both the MTA and T90 to improve performance. Tuning on the MTA was done primarily on a single processor to achieve effective multithreading, whereas tuning on the T90 was done primarily on multiple processors to achieve effective multitasking. In some instances the T90 tuning resulted in lower single-processor performance.

Three levels of tuning were applied on the MTA (beyond the initial, untuned level):

Level 0: no tuning;
Level 1: minimal tuning to get major loops parallel (e.g., by inserting pragmas);
Level 2: standard tuning;
Level 3: heroic tuning, which requires deep insight into algorithm and/or architecture.

Level 1 was conducted without feedback from actual timings. Levels 2 and 3 involve the standard feedback loop of profiling, locating bottlenecks, and reprogramming. Introducing these levels allows quantification of the effort spent in tuning and the performance gains achieved relative to the effort expended.

All five kernels were tuned to Level 3, heroic tuning, on the MTA. Somewhat less tuning effort was invested on the T90 than on the MTA. Given the more mature software environment on the T90, the final tuned results are probably reasonably comparable in terms of the ultimate performance achievable on each machine for a particular kernel.

Table B-2 shows the performance of the five kernels of Class A size on the T90 and MTA at the two tuning extremes (except for one Class W case). Tuned results are also presented for two different MTA configurations – with 4 and 8 processors – and for the Class B size in several cases. Discussion of these results follows, kernel by kernel. Early versions of these results are presented in References [B-2] to [B-5].

Table B-2. Results for NPB 2.3-serial kernels

Kernel & size	Computer & tuning	Processors used/total	Speed (Mflops or Mops*)	Speedup	Efficiency
CG Class A	T90 untuned	1/	178		
	T90 tuned	1/	173	1.00	1.00
		2/	291	1.68	0.84
		4/	568	3.28	0.82
		8/	960	5.55	0.69
	MTA untuned	1/ 4	127		
	MTA tuned	1/ 4	171	1.00	1.00
		2/ 4	331	1.94	0.97
		4/ 4	596	3.49	0.87
		1/ 8	164	1.00	1.00
		4/ 8	619	3.77	0.94
		8/ 8	939	5.73	0.72
CG Class B	T90 tuned	1/	202	1.00	1.00
		2/	337	1.67	0.83
		4/	639	3.16	0.79
		8/	1,201	5.95	0.74
	MTA tuned	1/ 8	174	1.00	1.00
		8/ 8	1,201	6.90	0.86
EP Class A	T90 untuned	1/	6.84 *		
	T90 tuned	1/	6.84 *	1.00	1.00
		2/	13.71 *	2.00	1.00
		4/	27.36 *	4.00	1.00
		8/	54.61 *	7.98	1.00
	MTA untuned	1/ 4	1.68 *		
	MTA tuned	1/ 4	7.57 *	1.00	1.00
		2/ 4	14.85 *	1.96	0.98
		4/ 4	29.50 *	3.90	0.97
		1/ 8	7.57 *	1.00	1.00
		4/ 8	30.38 *	4.01	1.00
		8/ 8	61.02 *	8.06	1.01
EP Class B	MTA tuned	1/ 8	7.51 *	1.00	1.00
		8/ 8	60.35 *	8.04	1.00

Table B-2. Results for NPB 2.3-serial kernels (continued)

Kernel & size	Computer & tuning	Processors used/total	Speed (Mflops or Mops*)	Speedup	Efficiency
FT Class A	T90 untuned	1/	164		
	T90 tuned	1/	696	1.00	1.00
		2/	1,312	1.89	0.94
		4/	2,343	3.37	0.84
		8/	3,768	5.41	0.68
FT Class W	MTA untuned	1/ 4	28		
FT Class A	MTA tuned	1/ 4	187	1.00	1.00
		2/ 4	350	1.87	0.94
		4/ 4	701	3.75	0.94
		1/ 8	187	1.00	1.00
		4/ 8	754	4.03	1.01
		8/ 8	1,421	7.60	0.95
FT Class B	MTA tuned	1/ 8	187	1.00	1.00
		8/ 8	1,408	7.53	0.94
IS Class A	T90 untuned	1/	74 *		
	T90 tuned for 4p	1/	42 *	1.00	1.00
		2/	81 *	1.93	0.96
		4/	164 *	3.90	0.98
	T90 tuned for 8p	8/	160 *		
	MTA untuned	1/ 4	61 *		
	MTA tuned	1/ 4	110 *	1.00	1.00
		2/ 4	214 *	1.95	0.97
		4/ 4	366 *	3.33	0.83
		1/ 8	112 *	1.00	1.00
		4/ 8	421 *	3.76	0.94
		8/ 8	699 *	6.24	0.78
IS Class B	MTA tuned	1/ 8	112 *	1.00	1.00
		4/ 8	440 *	3.93	0.98
		8/ 8	724 *	6.46	0.81
MG Class A	T90 untuned	1/	563		
	T90 tuned	1/	556	1.00	1.00
		2/	1,106	1.99	0.99
		4/	2,170	3.90	0.98
		8/	3,860	6.94	0.87
	MTA untuned	1/ 4	138		
	MTA tuned	1/ 4	184	1.00	1.00
		2/ 4	369	2.01	1.00
		4/ 4	702	3.82	0.95
		1/ 8	194	1.00	1.00
		4/ 8	738	3.80	0.95
		8/ 8	1,262	6.51	0.81
MG Class B	MTA tuned	1/ 8	208	1.00	1.00
		8/ 8	1,352	6.50	0.81

CG (Conjugate Gradient)

Tuned CG Class A performance is nearly the same on one processor of either the T90 or MTA. Tuning increased MTA performance from 127 to 171 Mflops (on the 4-processor machine), whereas T90 performance decreased slightly from 178 to 173 Mflops on one processor when multitasking was implemented. On multiple processors, the MTA initially scales this kernel better than the T90, but by 8 processors the performance is again nearly the same on either machine.

The inner loop of CG does three memory references and two floating-point operations. Because the MTA can only do one memory reference per cycle, any tuning that can increase the ratio of floating-point operations to memory operations will have a significant impact. One of three memory values read per loop iteration is an index value with a known upper bound. Knowing this upper bound, it is possible to pack three index values into a single 64-bit word in the preprocessing phase of CG (which is not timed when reporting performance results). This reduces the ratio of memory references to floating-point operations from 3/2 to 7/6 and should lead to a performance improvement of a factor of 1.29, which is in good agreement with the observed factor of 1.35 (=171/127). In fact, there is an added benefit to reducing the memory intensity of a loop; contention on the memory network is reduced, and thus effective memory latencies are shortened.

Single-processor tuning of CG on the MTA can be summarized as follows:

Level 0: CG, out of the box, achieved 127 Mflops on one processor of the MTA. The inner loop is highly efficient: two flops per three instructions with three memory references. This is as efficient as one could expect with the inner loop expression of sum = sum + a(k)*p(colidx(k)), which is exactly two floating-point operations and three memory accesses counting the indirection.

Level 1: No changes were required to the source to make the compiler parallelize the major loops.

Level 2: No changes at this level were attempted.

Level 3: As described before, bit packing increased the speed to 171 Mflops.

Table B-2 contains Class A results on two MTA configurations: the 4-processor system used for tuning and the 8-processor system that became available late in the project. Scalability is as good or better for all kernels on the larger system. This is presumably due to a combination of greater aggregate memory bandwidth and an improved compiler on the larger system. For CG the parallel efficiency at 4 processors increased from 87% to 94% going from the 4-processor to the 8-processor system. At the same time the single-processor speed decreased by 4%, possibly due to an offsetting compiler change.

Table B-2 also contains Class B results on 8 processors of both the T90 and MTA. Scalability is much better for CG on the larger Class B problem, with the efficiency at 8 processors improving from 72% to 86%. The improvement in Class B scalability on the T90 is smaller, but the single-processor speed is higher, so the final 8-processor speeds are the same on both machines.

EP (Embarrassingly Parallel)

EP performs well and scales nearly perfectly on both the T90 and MTA after tuning. Absolute performance is 11% better on the MTA than on the T90 on both one and eight processors. To obtain multithreaded code on the MTA, it is necessary to use the shared-memory version of the random number generator that comes with the NPB distribution.

Single-processor tuning of EP on the MTA can be summarized as follows:

Level 0: When the study began, the outer loop did not parallelize, so the untuned code ran in serial and very slowly, at 0.12 Mops. Subsequent improvements to the compiler allowed the outer loop to parallelize, resulting in a dramatic increase in speed to 1.68 Mops.

Level 1: Five lines were changed. The original code computed a vector of random numbers with each call to the generator. Separate storage needs to be allocated for each thread in a shared-memory parallelization. This was accomplished with two compiler directives and three modifications to declarations. Performance then increased to 2.39 Mops.

Level 2: Changes to five more lines of code eliminated a memory hot spot: a global variable location that multiple threads update. Since it is not logically a synchronization point, replication was used to relieve contention and increase performance by more than a factor of 2.

Level 3: Inlining some functions boosted performance to 7.57 Mops. Inlining might be considered Level 2, but assistance was provided by expert Tera staff, so this is counted as "heroic" tuning.

Going from the 4-processor to the 8-processor MTA, the efficiency improved by 3% on 4 processors. No increase in efficiency was observed in going from Class A to Class B.

FT (Fourier Transform)

FT is highly vectorizable, so the T90 outperforms the MTA by about a factor of 3.7 for tuned code on a single processor: 696 Mflops versus 187 Mflops. Scalability on the MTA is better than on the T90, so at 8 processors the T90 is only 2.7 times faster than the MTA.

The original version of the code had only 16-way outer-loop parallelism. Tuning for the MTA involved modifying the code to do multiple butterflies in parallel. Single-processor tuning of FT on the MTA can be summarized as follows:

Level 0: With no lines changed, the code encountered an "alignment error" and failed to complete for the Class A problem size. However, the smaller Class W size did run correctly, with a speed of 28 Mflops.

Level 1: No changes were required to make major loops parallel (again, for the Class W size).

Level 2: Loops were removed that copied 16 1-D transforms at a time into contiguous storage before performing vector transforms. Such copying is beneficial on a vector computer, but is pure overhead on the MTA. With these loops removed, the Class A size executed correctly, and performance improved to 118 Mflops.

Level 3: By restructuring loops to expose more outer loop parallelism, performance increased to 187 Mflops.

When the tuned code was run on 4 processors of the 8-processor MTA, the efficiency improved to 101%, as compared to 94% on the smaller 4-processor system. No significant change in MTA speed was noted in going from Class A to Class B size.

IS (Integer Sort)

IS makes use of memory indirection. The MTA handles this well, achieving a single-processor speed of 110 Mops after tuning. Without tuning, the T90 does a credible job of vectorization to give 74 Mops on a single processor.

IS scales reasonably well on the MTA, but not on the T90. To parallelize IS on the T90 a work array was duplicated on each processor and hard-coded for a particular number of processors. This increased the work, leading to a significant drop in speed on one processor and only a net speedup of 2.3 on four processors over the untuned value. The implementation for 8 T90 processors showed no further improvement over the 4-processor version. Overall, the MTA is the clear winner and more than 4 times faster than the T90 at 8 processors.

Single-processor tuning of IS on the MTA can be summarized as follows:

Level 0: With no lines changed the speed was 61 Mops.

Level 1: No changes were needed to make problem parallel.

Level 2: Removing some copy loops that were not beneficial to a shared-memory machine required changing five lines of code. Performance improved to 73 Mops.

Level 3: Bitpacking, similar to the heroic optimization of CG, increased the performance to 112 Mops.

When the tuned code was run on 4 MTA processors, the efficiency increased from 83% to 94% after the MTA was upgraded from 4 to 8 processors. This shows the importance of additional memory bandwidth for a memory-intensive code such as IS. When the problem size increased from Class A to Class B, a modest 3% performance improvement was noted on 8 MTA processors.

MG (Multigrid)

MG is similar to FT in that it is highly vectorizable and very suitable to the T90. Accordingly, the T90 outperforms the MTA by a factor of 3 for tuned code on a single processor: 556 Mflops versus 184 Mflops. Scalability on both the T90 and MTA is similar, so the T90 is still a factor of 3 faster than the MTA at 8 processors.

Tuning for the MTA involved code restructuring to avoid register spills. Each thread on the MTA has access to only 32 floating-point registers so efficient register allocation is important. Single-processor tuning of MG on the MTA can be summarized as follows:

Level 0: With no lines changed the speed was 138 Mflops.

Level 1: No changes were required to make major loops parallel.

Level 2: Manually fusing loops that perform a 27-point stencil computation resulted in register spillage. A lot of effort was expended here without achieving a performance gain.

Level 3: Further code manipulation by expert Tera staff fused some loops and eliminated some data copying to increase the speed to 184 Mflops.

For the tuned code run on 4 MTA processors, no improvement in efficiency was observed going from the 4-processor to the 8-processor system. However, the absolute performance improved slightly, by 5%. When the problem size increased from Class A to Class B, performance improved by 7% on 8 MTA processors.

Conclusions

For each Class A kernel, Figure B-1 shows the tuned speed per processor on the T90 and MTA at 8 processors normalized to the best single-processor T90 speed. For the T90 this is just the parallel efficiency, which is reasonably good for all of the kernels except IS.

Comparing the MTA with the T90 at 8 processors, the two machines are essentially the same speed for CG and EP; the T90 is significantly faster for FT and MG; and the MTA is significantly faster for IS. As regards efficiency at 8 processors, it is better on the MTA for CG, FT, and IS, about the same on each machine for EP, and better on the T90 for MG.

The MTA cannot outperform the T90 processor to processor on codes with long vectors. However, the MTA can perform respectably on these codes and scale them well. The MTA can also exploit parallelism at higher levels to achieve performance on codes that do not perform well on the T90. The MTA thus offers a wider range of parallel options than the T90.

The upgrade of the MTA from 4 to 8 processors more than doubled its aggregate memory bandwidth. This, along with associated compiler changes, improved the 4-processor performance of all of the kernels on the upgraded machine compared to the smaller machine. The improvement was most dramatic for IS, which is highly memory intensive. This shows the importance of memory bandwidth on the MTA. It further suggests that memory bandwidth may be critical for scaling memory-intensive codes on large MTA configurations.

In another set of tests, the problem size was increased from Class A to Class B on the 8-processor MTA. This improved scalability for two of the kernels and absolute performance for four of the kernels.

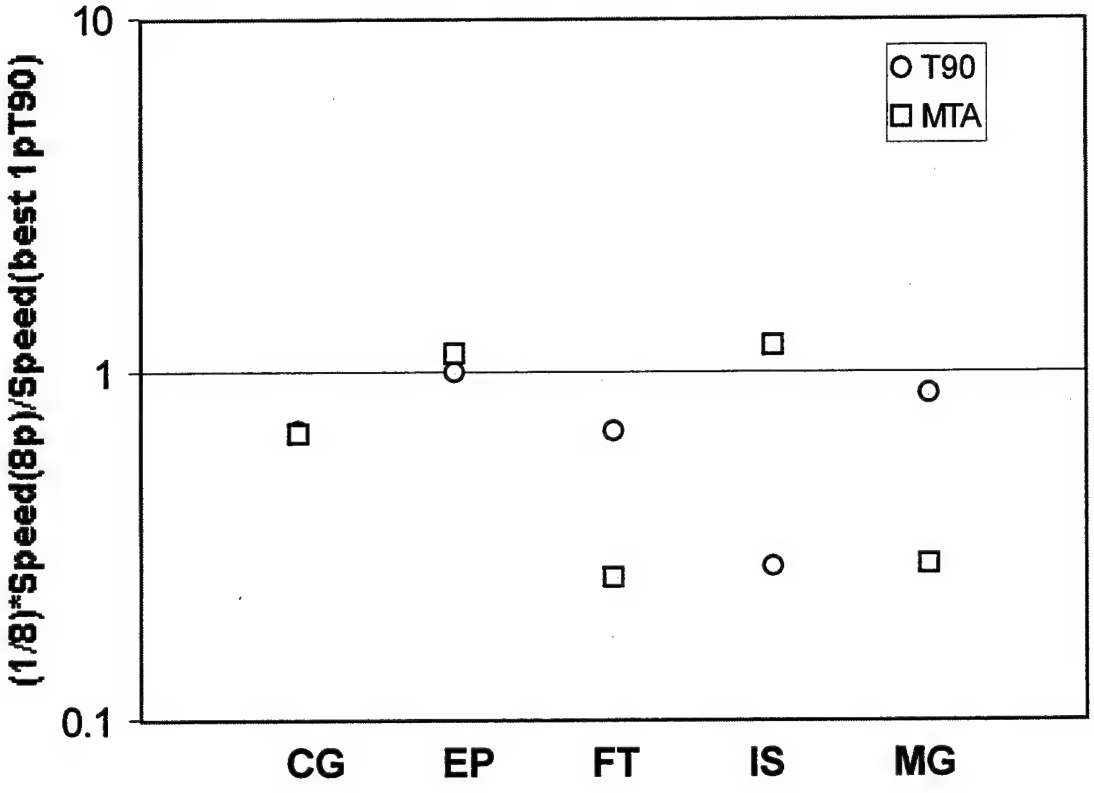


Figure B-1. Tuned speed per processor for each machine at 8 processors normalized to best single-processor T90 speed, all for Class A kernels.

The MTA was found to be as easy to program as the T90 and much easier to program than distributed-memory machines. The MTA is able to exploit inner-loop (vector) parallelism but prefers outer-loop parallelism. Indeed, a good tuning strategy on the MTA is to find parallelism and move it to an outer loop.

References

B-1. www.nas.nasa.gov/Software/NPB.

B-2. J. Boisseau, L. Carter, K. Gatlin, A. Majumdar, and A. Snavely, "NAS Benchmarks on the Tera MTA," *Proceedings of Workshop on Multi-Threaded Execution, Architecture, and Compilers,* Las Vegas NV (January 1998).

B-3. J. Boisseau, L. Carter, A. Snavely, D. Callahan, J. Feo, S. Kahan, and Z. Wu, "Cray T90 vs. Tera MTA: The Old Champ Faces a New Challenger," *Proceedings of Cray User Group Conference,* Stuttgart, Germany (June 1998).

B-4. A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K.S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-Processor Performance on the Tera MTA," *Proceedings of SC98,* Orlando FL (November 1998).

B-5. L. Carter, J. Feo, and A. Snavely, "Performance and Programming Experience on the Tera MTA," *Proceedings of Ninth SIAM Conference on Parallel Processing for Scientific Computing,* San Antonio TX (March 1999).

TPHOT – Monte Carlo Photon Transport

Amit Majumdar
San Diego Supercomputer Center

Introduction

TPHOT is a time-dependent Monte Carlo code that simulates photon transport in a plasma. Various versions of THPOT have been used for several years to examine and benchmark parallel Monte Carlo particle transport on a variety of computers [C-1 to C-4].

TPHOT has been implemented and parallelized on the Tera MTA and Cray T3E at SDSC using different parallelization strategies. The code and strategies are described here, along with the results obtained on up to 8 MTA processors and 64 T3E processors.

Description of TPHOT

Monte Carlo particle transport simulates the behavior of particles by drawing random samples from appropriate probability distributions. A source particle is generated from a known distribution with a random position and velocity (or energy and direction for a photon). Given these particle characteristics and the properties of the medium in which the particle is traveling, transport is simulated by sampling from additional probability distributions describing the interaction of the particle with the medium. The particle moves in a straight line between interactions, and the simulation proceeds until the particle is absorbed or escapes.

TPHOT simulates photon transport within a high-density, high-temperature plasma in two-dimensional r-z geometry. The plasma is divided into material zones, each with its own composition, temperature, and density. Each zone is a simple volume of revolution, bounded by at most four surfaces.

Photons are sampled uniformly and isotropically within each zone from a Planckian energy spectrum. The energy range is discretized into several energy groups. The photons that are emitted within each zone and energy group are followed through the plasma until they are absorbed, undergo Thompson scattering, escape, or reach "census" at the end of a time step. The sequence of movements and interactions of each photon during a time step is referred to as its "history".

In TPHOT, the geometry and material properties are constant in time, and time stepping is used only to determine when results are output to a census file. In a more general case, one might model coupled photon transport and hydrodynamics, in which case the geometry and material properties could change from one time step to another.

The photon transport computation is a triply-nested loop over zones, energy groups, and photons. Relevant pseudo-code for these loops follows.

```
do i = 1, number_of_zones
  do j =  1, number_of_energy_groups
      .
      .
    number_of_photons_in_i_and_j = function of (edep)
    do k = 1, number_of_photons_in_i_and_j
      call multiple subroutines to track the photon history
    end do
  end do
end do
```

The variable `edep`, which is the energy deposited in each zone and energy group, is a function of the Planckian energy spectrum, the material properties, and the volume of each zone.

Since the photons do not interact, their histories are independent and can be computed simultaneously. This allows an embarrassingly parallel implementation, which should exhibit nearly perfect linear speedup, provided that 1) the workload can be balanced by a suitable assignment of photons to processors, and 2) each processor has ready access to the material properties of all the zones (which requires that the zone data fit in local memory on a distributed-memory computer). To obtain reproducible results, each processor should also generate a non-overlapping, independent sequence of random numbers.

Parallelization by Zones on the MTA

Since the computations across zones are independent and the Tera MTA prefers outer-loop parallelism, the first implementation on the MTA parallelized only the outermost zone loop. The inner loops include many subroutine calls and shared accumulators, so the Tera compiler was not able to parallellize the outermost loop automatically. Thus an `ASSERT PARALLEL` directive and several `ASSERT LOCAL` directives were inserted to identify the local variables. Moreover, each global tally (for the number of photons escaped, number of photons absorbed, etc.) had to be preceded with an `UPDATE` directive to insure determinacy. Insertion of the directives was time consuming and error prone. The parallel pseudo-code, including the aforementioned directives in bold, is as follows.

```
C$TERA ASSERT PARALLEL
      do i = 1, number_of_zones
C$TERA ASSERT LOCAL (names of local variables)
        do j =  1, number_of_energy_groups
            .
            .
          number_of_photons_in_i_and_j = function of (edep)
          do k = 1, number_of_photons_in_i_and_j
            call multiple subroutines to track the photon history
            (C$TERA UPDATE directive before each tally statement)
          end do
        end do
      end do
```

Parallelization by Zones and Energies on the MTA

As will be seen shortly, parallelization over zones only does not scale well on the MTA for the test problem. Evidently, there is insufficient parallelism to get good load balance. Hence parallelism was increased by collapsing the two outer loops over zones and energies into a single loop. Also, the order of processing the zones was reversed, since the number of photons in a zone is proportional to its size, and the size of the zones grows with index value for the test problem. The resulting pseudo-code is as follows.

```
C$TERA ASSERT PARALLEL
      do ij = number_of_zones * number_of_energy_groups - 1, 0, -1
C$TERA ASSERT LOCAL (names of local variables)
            .
            .
          i = (ij/ number_of_energy_groups) + 1
          j = ij - number_of_energy_groups*(i - 1) + 1
            .
          number_of_photons_in_ij = function of (edep)
          do k = 1, number_of_photons_in_ij
            call multiple subroutines to track the photon history
            (C$TERA UPDATE directives before each tally statement)
          end do
        end do
      end do
```

Parallelization by Photons on the T3E

The parallel version of TPHOT developed for the Cray T3E uses the MPI library, which is appropriate for distributed-memory machines such as the T3E. Moreover, the parallelization strategy was different from that used on the MTA. If NP is the number of processors available, then the parallel pseudo-code for the computation-intensive loops is as follows, with the modified part of the pseudo-code in bold letters.

```
do i = 1, number_of_zones
  do j =  1, number_of_energy_groups
      .
      .
    number_of_photons_in_i_and_j = function of (edep/NP)
    do k = 1, number_of_photons_in_i_and_j
      call multiple subroutines to track the photon history
    end do
  end do
end do
.
.
call MPI_REDUCE(...) to add up tally variables.
```

This code is executed by each processor. The net effect of dividing `edep` by NP is that each of the NP processors simulates 1/NP of the total number of photons. This effectively parallelizes across the number of photons.

At the end of the simulation one of the processors needs to perform multiple reduction operations (such as the `MPI_REDUCE` summation operation) to add up various global tallies (for the number of photons escaped, number of photons absorbed, etc.). Besides these reduction operations at the end, no other MPI library calls are required other than the MPI initialization calls at the beginning to identify the total number of processors, each processor's MPI rank, etc.

All the processors execute the initial part of the code where they read input parameters, assign material properties to zones, etc. As noted previously, one requirement for this parallelization strategy is that the storage for all the zones must fit in the local memory of each processor.

Results

The physical problem simulated is transport through an inertial confinement fusion plasma consisting of a 50%-50% mixture of deuterium and tritium surrounded by SiO_2, all at elevated temperature and density. A single time step is modeled, during which approximately 24,000,000 photons are emitted. The material is divided up into 1,960 zones, arising from 49 axial mesh intervals and 40 radial mesh intervals. Twelve energy groups are used. For output the code keeps track of the number of photons absorbed and the number of photons escaping the plasma, as well as the number of photons escaping in each energy group.

Table C-1 gives performance results for solving the test problem with TPHOT on the MTA and T3E. The listed times do not include input and output. On a single-processor, the MTA is about four times faster than the T3E. Scalability on the T3E is nearly linear to 64 processors. Scalability on the MTA is poor when parallelization is only by zones, but nearly linear to 8 processors when parallelization is by both zones and energies.

Parallelization on the T3E is done across the 24,000,000 photons. Even on 64 processors, the work per processor is substantial. Since the computation is embarrassingly parallel and there is little communication between processors, scalability is linear. The poor scalability on the MTA when parallelized across just the 1,960 zones is due to insufficient parallelism. Parallelizing over both zones and energy groups provides sufficient parallelism to cover latencies, amortize overheads, and load balance the work on each stream.

For this embarrassingly parallel problem, the distributed memory of the T3E is actually an advantage. Information for each particle is naturally kept separate on different processors. The only need for communication is for tallying system values (MPI_REDUCE) at the end of the time step. By contrast, on the MTA with its shared memory, it is necessary to avoid having different threads update shared variables in an irreproducible manner. Thus, all common variables, such as the positions and angles, need to be identified and declared as local. In addition, the atomicity of tally operations needs to be insured by inserting pragmas before each operation.

The good scalability on the T3E is made possible because the data for all the zones fit in the memory of each local processor. For much larger problems this would not be possible. A different parallel implementation would then be needed and would likely not scale as well. On the MTA, on the other hand, the only issue is parallelism. As long as, the number of zones times the number of energy groups is much larger than the number streams, performance will scale linearly.

If more parallelism were needed on the MTA, the parallelization could, in fact, be done across photons. One way would be to add a loop over threads outside the loop over zones. If the total number of threads to be used is NT, then `edep` would be divided by NT (instead of by NP, as in the MPI code), and each thread would simulate 1/NT of the total photons. Some existing declarations would have to be moved, and a few new ones would need to be added.

Table C-1. Performance results for TPHOT on the MTA and T3E

Computer & strategy	Processors	Time (s)	Speedup	Efficiency
MTA by zones	1	737	1.00	1.00
	2	386	1.91	0.95
	4	219	3.37	0.84
	8	161	4.58	0.57
MTA by zones & energies	1	718	1.00	1.00
	2	356	2.02	1.01
	4	178	4.03	1.01
	8	91	7.88	0.99
T3E by photons	1	3,008	1.00	1.00
	2	1,506	2.00	1.00
	4	757	3.97	0.99
	8	378	7.96	0.99
	16	190	15.8	0.99
	32	95	31.7	0.99
	64	48	62.7	0.98

References

C-1. F. W. Bobrowicz, J. E. Lynch, K. J. Fisher, and J. E. Tabor, "Vectorized Monte Carlo Photon Transport,'" *Parallel Computing*, 1, 295-305 (1984).

C-2. W. R. Martin, P. F. Nowak, and J. A. Rathkopf, "Monte Carlo Photon Transport on a Vector Supercomputer," *IBM Journal of Research and Development*, 30, 193 (1986).

C-3. W. R. Martin, T. C. Wan, T. S. Abdel-Rahman, and T. N. Mudge, "Monte Carlo Photon Transport on Shared Memory and Distributed Memory Parallel Processors," *International Journal of Supercomputer Applications*, 1 (3), 57 (1987).

C-4. W. R. Martin, A. Majumdar, J. A. Rathkopf, and M. Litvin, "Experiences with Different Parallel Programming Paradigms for Monte Carlo Particle Transport Leads to a Portable Toolkit for Parallel Monte Carlo," *Proceedings of International Joint Conference on Mathematical Methods and Supercomputing in Nuclear Applications*, Karlsruhe, Germany, Vol. II, 418 (April 1993).

C3I Parallel Benchmark Suite

Sharon Brunett, John Thornley, and Marrq Ellenbecker
California Institute of Technology

John Feo
Tera Computer

Performance for two C3I benchmarks has been investigated on the MTA and compared to that on conventional uniprocessor and multiprocessor architectures. This case study summarizes the performance obtained and the associated programming effort. Preliminary results were reported in Ref. [D-1].

Benchmark Problems

The U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite [D-2] consists of eight problems that compactly represent the essential elements of real C3I applications. Each problem consists of the following: 1) a problem description giving the inputs and required outputs; 2) an efficient sequential program written in C to solve the problem; 3) the benchmark input data; and 4) a correctness test for the benchmark output data. The chosen C3IPBS problems – Threat Analysis and Terrain Masking – are computationally intensive, memory intensive, and compact; they involve nontrivial data and control structures; and they have the potential for large-scale parallelization.

Experimental Goals

Using the Threat Analysis and Terrain Masking benchmarks, initial answers to the following questions were explored:

1. What is the performance of a multithreaded architecture, such as the Tera MTA, compared to conventional uniprocessor and multiprocessor architectures?

2. What are appropriate methods for developing efficient, general-purpose programs on a multithreaded architecture, and how much programmer effort is required?

3. What are the difficulties in ensuring sufficiently large numbers of threads are available to saturate processors?

Experiments were performed on conventional uniprocessor and multiprocessor architectures, as well as on the Tera MTA. Table D-1 gives the computers used in the performance comparisons. Also listed is the HP V2250, which was used in the thermal explosion case study described in a subsequent section. For the X2000 the number of processors listed is the maximum actually used (64), which is smaller than the total number available (256).

Table D-1. Computers used in performance comparisons

Computer	Processors	Operating system
Digital AlphaStation	1 x 500-MHz Digital Alpha 21164A	Digital Unix 4.0C
NeTpower Sparta	4 x 200-MHz Intel Pentium Pro	Windows NT 4.0
HP Exemplar X2000	64 x 180-MHz HP PA-8000	SPP-UX 5.3
HP V2250	32 x 240-MHz HP PA-8200	HP-UX 11.01
Tera MTA	8 x 260-MHz custom GaAs	MT Unix

Threat Analysis

The Threat Analysis problem is a time-stepped simulation of the trajectories of incoming ballistic threats with computation of options for intercepting the threats. The input to the problem consists of (i) the trajectories of a set of incoming threats, and (ii) the locations and capabilities of a set of weapons that can be used to intercept the incoming threats. For each threat and weapon pair, the program must compute the time intervals over which the threat can be intercepted by the weapon. The program computes a set of tuples of the form (threat, weapon, interval) indicating that the weapon can intercept the threat over a particular time interval. Because of the constraints on threat interception, there can be zero, one, or more intervals associated with each (threat, weapon) pair.

Automatic Parallelization. On both the HP Exemplar and Tera MTA, the manufacturer-supplied automatic parallelizing compilers were unable to identify any practical opportunities for parallelizing the sequential Threat Analysis program. The compilers and analysis tools were unable to suggest changes to the program (e.g., algorithmic modifications or the addition of pragmas) that might expose parallelism. The reason is that the algorithm is inherently sequential. The outer-loop iterations assign to shared variables, and the inner loops are sequential time-stepped simulations. In addition, the program (like most general-purpose programs) contains chains of function calls, pointer operations, and non-trivial index expressions, which thwart compiler analysis and make automatic parallelization extremely difficult.

Multithreaded Solution. The program, however, can be manually parallelized through relatively straightforward algorithmic modifications as follows. The outer loop over all threats is replaced by a multithreaded loop in which each iteration is responsible for a different chunk (i.e., subrange) of the threats. The problem of the shared variables is solved by modifying the algorithm so that each iteration increments a private counter and assigns its own contribution to the solution, interval array. Moving those variables to the inner blocks localizes declarations of other variables. The reworked code has outer-loop iterations that are completely independent of each other and able to be executed by separate threads. Lastly, a compiler pragma above the revised outer loop is necessary to instruct the compiler to run the loop in parallel.

A drawback of this multithreaded solution is the necessity for a larger interval array than was needed in the sequential program. Since there is no way to determine in advance the number of intervals that each iteration will compute, each iteration's section of the interval array must be generously oversized. Therefore, the larger the number of chunks, the larger the interval array. Parallelism is achieved at the expense of extra memory requirements.

Performance Comparison of Sequential and Multithreaded Threat Analysis. Table D-2 gives the sequential and multithreaded total execution times for five input scenarios of the Threat Analysis benchmark on the computers used in this evaluation. The untuned MTA time is sequential, while the other MTA times are multithreaded.

The single-processor performance on the three conventional computers varies roughly in proportion to the respective processor speeds. The benchmark program is compute-bound, rather than memory-bound, so the faster processors perform better. By contrast, the single-processor sequential speed of the MTA is extremely slow – roughly 13 times slower than the Alpha and nine times slower than a single X2000 processor. The MTA is not efficient for execution of single-threaded (sequential) programs. The principal reason is that a single thread on the Tera MTA can issue only one instruction every 21 cycles, giving at most 5% processor utilization.

Multithreaded speedups on the Pentium Pro computer are excellent up to the four processors available. Threads are completely independent and execute mostly within cache. The program was manually parallelized using the Caltech Sthreads library [D-3] implemented on top of the Win32 thread API [D-4] supported by Windows NT and was executed using one chunk/thread per Pentium Pro processor. Similarly, good speedups are achieved on the X2000 up to more than 32 processors, again because threads are completely independent and execute mostly within cache.

Multithreaded performance on a single MTA processor is dramatically faster, by a factor of 30, compared to sequential performance and a factor of two or more faster than on any of the other single processors. However, speedup on multiple MTA processors is poor and peaks at only seven

Table D-2. Performance results for Threat Analysis

Computer	Processors	Time (s)	Speedup	Efficiency
Digital Alpha	1	187		
Pentium Pro	1	458	1.00	1.00
	4	117	3.91	0.98
HP Exemplar X2000	1	280.1	1.00	1.00
	2	140.5	1.99	1.00
	4	70.54	3.98	0.99
	8	35.40	7.91	0.99
	16	18.18	15.4	0.96
	32	9.39	29.8	0.93
	64	5.64	49.7	0.78
Tera MTA untuned	1	2,485		
Tera MTA	1	83.4	1.00	1.00
	2	48.2	1.73	0.87
	3	32.6	2.56	0.85
	4	26.4	3.16	0.79
	5	23.4	3.56	0.71
	6	21.3	3.92	0.65
	7	20.8	4.01	0.57
	8	20.9	3.99	0.50

processors, resulting in a speed there that is slightly slower than for 16 X2000 processors. Presumably the 1,000 threats in each benchmark scenario are insufficient to achieve good processor utilization for more than a few MTA processors. This demonstrates that more parallelism is needed to achieve good speedup on the MTA than on conventional multiprocessors.

Terrain Masking

The Terrain Masking problem is a computation of the maximum safe flight altitude over all points in an uneven terrain containing ground-based threats. The input to the problem consists of (i) the ground elevation for all points in the terrain, and (ii) the position and range of the threats. The output of the problem consists of the maximum altitude at which an aircraft is invisible to all threats for all points in the terrain.

For each threat in turn, the benchmark program computes the maximum safe flight altitudes due to the threat over its region of influence, then minimizes these altitudes into the overall result. The maximum safe flight altitudes due to a threat cannot be computed directly into the overall result because the value at one point is computed from the values at neighboring points.

Automatic Parallelization. As was the case for the previous program, the HP and Tera compilers were not able to identify any meaningful opportunities for automatic parallelization of the sequential Terrain Masking program. Moreover, the compilers and analysis tools were unable to suggest changes to the program that might expose parallelism. The outer loop of the program cannot be parallelized without algorithmic modifications, since its iterations over all threats assign to overlapping regions of the solution, masking array. The inner loops contain opportunities for parallelization. Similar to the previous program, the Terrain Masking program contains chains of function calls, pointer operations, and non-trivial index expressions that thwart compiler analysis and make automatic parallelization extremely difficult.

Multithreaded Solutions. The outer loop over all threats is not immediately parallelizable, because the regions of influence of different threats can overlap. A straightforward parallelization solution requires that a locking scheme be used for access to the masking array, to ensure that multiple threads do not assign to overlapping regions of masking.

The outer loop over all threats is replaced by a multithreaded loop inside which each iteration dynamically processes individual threats until all threats have been processed. The problem of shared access to the masking array is solved by blocking the terrain into equal-sized blocks, with a separate lock associated with each block. The role of the temp and masking arrays is swapped in the computation of the maximum safe flight altitudes in the region of influence of a threat. The temp array is minimized back into the masking array block by block. To avoid interference between threads, blocks are locked before writing and unlocked after writing.

The principal drawback of the preceding coarse-grained implementation is that each thread requires its own temp array. For the benchmark problem the region of influence of each threat is up to 5% of the total terrain. Therefore, this approach does not require excessive extra storage for small numbers of threads (e.g., 16), but is impractical for the large numbers of threads (e.g., hundreds) needed on multiple processors of the MTA.

Indeed, a finer grained implementation is necessary on the MTA. Such an approach parallelizes the inner loops that compute the maximum safe flight altitude for an individual threat. This is done using Tera parallelization pragmas and futures constructs. The fine-grained parallelization is neither easier nor more difficult than the coarse-grained parallelization, but is viable only for the MTA, not the conventional multiprocessors.

Performance of Sequential and Multithreaded Terrain Masking. Table D-3 gives the sequential and multithreaded total execution times for five input scenarios of the Terrain Masking benchmark on the computers used in this evaluation. The untuned MTA time is sequential, while the other MTA times are multithreaded.

Table D-3. Performance results for Terrain Masking

Computer	Processors	Time (s)	Speedup	Efficiency
Digital Alpha	1	158		
Pentium Pro	1	197	1.00	1.00
	4	65	3.03	0.76
HP Exemplar X2000	1	201.6	1.00	1.00
	2	112.8	1.79	0.89
	4	59.5	3.39	0.85
	8	37.3	5.40	0.68
	12	33.5	6.02	0.50
	16	37.1	5.43	0.34
Tera MTA untuned	1	978		
Tera MTA	1	37.1	1.00	1.00
	2	19.8	1.87	0.94
	3	12.7	2.92	0.97
	4	9.6	3.86	0.97
	5	7.9	4.70	0.94
	6	6.8	5.46	0.91
	7	6.2	5.98	0.85
	8	6.5	5.71	0.71

The single-processor performance on the three conventional computers varies relatively little between each other. This is because the program is memory-bound, rather than compute-bound, so processor speed is not the major determinant of execution time. As for the previous benchmark, the sequential speed on the MTA is much slower than on the other computers, by roughly a factor of six compared to the Alpha and a factor of five compared to the X2000. The speed difference is less than with Threat Analysis, because the conventional processors are not fully utilized in this memory-bound program.

Multithreaded speedups on the Pentium Pro and X2000 computers are poorer than for the previous benchmark. Indeed, the speedup on the X2000 peaks at 12 processors.

Multithreaded performance on a single MTA processor is again dramatically faster, by a factor of 26, compared to sequential performance and a factor of four or more faster than on any of the other single processors. Speedup is good up to five processors, but tapers off sharply thereafter and peaks at seven processors, where the speed is 5.4 times faster than on 12 X2000 processors. Again, insufficient parallelism is available to achieve good processor utilization on more than a few MTA processors.

Summary

Automatic Parallelization. On both the MTA and Exemplar, the manufacturer-supplied automatic parallelizing compilers were unable to identify any practical opportunities for parallelization in either of the two sequential benchmark programs. Nor were the compilers able to make any suggestions regarding changes to the program (e.g., algorithmic modifications, assertions, or pragmas) that might allow the program to parallelize.

Automatic parallelization of general-purpose programs is extremely difficult. There are two fundamental obstacles:

1. Efficient parallelization usually requires more than parallelization of loops in the sequential program. It involves significant modification of the underlying algorithm. This is the case with both benchmark programs. It is unreasonable to expect a compiler to deduce the high-level purpose of a program and then automatically develop an alternate algorithm to solve the same problem.

2. General-purpose programs typically involve hundreds of separately compiled modules, chains of function calls, non-trivial index expressions, and operations on pointers that thwart compiler analysis of data dependencies and program flow. With both benchmark problems, the compilers were not even able to parallelize the manually transformed programs without explicit parallel loop pragmas.

Manual Parallelization. The MTA and conventional coarse-grained multiprocessors have different strengths and weaknesses with regard to the ease of manual parallelization. These differences can be summarized as follows:

1. A weakness of the MTA is that it requires large numbers of threads for efficient execution. With the Threat Analysis program, splitting the outer loop into 16 threads yields over 15-fold speedup on a 16-processor Exemplar, whereas thousands of threads are evidently required for efficient execution on an 8-processor MTA.

   Splitting a program into many threads can be more difficult than splitting it into a few threads. The number of threads that can be obtained from the outer loop of the Terrain Masking problem is limited by the 60 threats per input scenario in the benchmark data sets. This provides plenty of threads for the Exemplar, but not enough for the MTA.

   Splitting a program into many threads can require more memory than splitting it into a few threads, because data replication is often proportional to the number of threads. For both benchmark problems, outer-loop parallelization requires extra array storage for each thread.

2. A strength of the MTA is that it provides hardware support for truly fine-grained multithreading. For both benchmark problems, algorithms based on fine-grained multithreading of inner loops are practical on the Tera MTA that are not practical on the conventional multiprocessors. This suggests that the MTA offers more options for parallelization.

   On conventional multiprocessors with operating system support for threads, thread creation costs tens of thousands to hundreds of thousands of cycles and thread synchronization

costs hundreds to thousands of cycles. On the MTA, thread creation and synchronization costs are lower.

Sequential Execution. The most serious performance weakness of the MTA is its extremely slow execution of sequential (single-threaded) programs. Sequential execution on the 260-MHz MTA is much slower than on any of the three conventional computers considered. For both C3I benchmark programs, sequential execution on the MTA is about five times slower than sequential execution on a 200-MHz Pentium Pro. The MTA is 13 times slower than a 500-MHz Alpha for the compute-bound Threat Analysis benchmark and six times slower for the memory-bound Terrain Masking benchmark. The principal reason for the MTA's poor performance with single-threaded programs is that a single thread can issue only one instruction every 21 cycles.

Poor single-threaded performance is clearly a practical problem for users porting sequential programs from other computers. It often takes a considerable amount of time to parallelize a sequential program, and the user may want some kind of acceptable performance while parallelization is in progress. In addition, some programs or parts of programs are inherently sequential.

Multithreaded Execution. Multithreaded programs with enough threads run dramatically faster than single-threaded programs on the MTA. For the two benchmark programs the single-processor MTA speeds improve by factors of 30 and 26 when multithreaded. As a result a single MTA processor is 3.4 times faster than a 180-MHz PA-8000 processor in the X2000 for the Threat Analysis benchmark and 5.4 times faster for the Terrain Masking benchmark.

However, speedup on the MTA is poor for both benchmarks and, in fact, peaks at seven processors. Evidently not enough parallelism is available for the MTA in either of the relatively small benchmarks. This is also a problem on the X2000 for the second benchmark, but not for the first. Accordingly, seven MTA processors are slightly slower than 16 X2000 processors for the Threat Analysis benchmark, but 5.4 times faster than 12 X2000 processors for the Terrain Masking benchmark. The results for the Threat Analysis benchmark demonstrate that more parallelism is needed to achieve good speedup on the MTA than on conventional multiprocessors.

References

D-1. Sharon Brunett, John Thornley, and Marrq Ellenbecker, "An Initial Evaluation of the Tera Multithreaded Architecture and Programming System Using the C3I Parallel Benchmark Suite," *Proceedings of SC98*, Orlando FL (November 1998).

D-2. Richard C. Metzger, Brian VanVoorst, Luiz S. Pires, Rakesh Jha, Wing Au, Minesh Amin, David A. Catanon, and Vipin Kumar, "The C3I Parallel Benchmark Suite - Introduction and Preliminary Results," *Proceedings of SC96*, Pittsburgh PA (November 1996).

D-3. John Thornley, K. Mani Chandy, and Hiroshi Ishii, "A System for Structured High-Performance Multithreaded Programming in Windows NT," *Proceedings 2nd USENIX Windows NT Symposium*, pp 67-76, Seattle WA (August 1998).

D-4. Jim Beveridge and Robert Wiener, Multithreading Applications in Win32: the Complete Guide to Threads, Addison-Wesley Developers Press, Reading MA (1997).

Explode – Thermal Explosion Benchmark

Sharon Brunett and Roy Williams
California Institute of Technology

Model Description

Explode – a thermal explosion benchmark – simulates an explosive wave that initiates chemical reactions in a reactive material. The code employs a fixed 2-D square mesh, but has a time-varying computational workload at each mesh point. In particular, the code solves differential equations describing thermal diffusion as well as temperature-sensitive reactions [E-1]. When the temperature rises, the time step for solving the reaction equations must become smaller. Thus an explosive wave moving through the mesh carries not only a high temperature, but also a high computational workload.

Parallelization and Tuning

Dynamic load balancing is required to achieve good performance in Explode and has proven difficult to implement effectively on conventional high-end computers, such as the HP X2000 and V2250. The possibility that the Tera MTA might be better suited for this code motivated the investigation described here.

Parallelization on all three computers used a shared-memory programming model. As with the C3I benchmarks, automatic parallelization by the Tera and HP compilers was thwarted by global variables within loops and function calls. Modifications to help the compilers parallelize safe code segments were mostly straightforward, but required understanding the code and its data dependencies. Tera and HP compiler directives, along with code reorganization, were necessary for initial parallel implementations. Since Explode was written from scratch, there was no sequential code to rework, as was the case with the C3I benchmarks.

Tuning on the MTA paid special attention to moving parallelism to the outer loopa. Also, the number of streams and dynamic scheduling were specified for certain loops. Such tuning was facilitated by use of the Traceview tool.

Tuning on the HP systems involved the use of process-monitoring tools to identify the performance inhibitors. As expected, memory contention and data cache misses from particular sections of the code were to blame. Straightforward fixes resolved some of these problems and resulted in significant performance improvements.

Although not attempted, a distributed-memory implementation of Explode on the HP systems would have required periodic data redistribution to keep all of the processors busy. This would have entailed communication and bookkeeping overhead whenever the load-balancing step was done, which was not necessary in the shared-memory implementation.

Test Problem and Performance

The test problem uses a square mesh with 301x301 mesh points. The diffusion time step is constant, while the reaction time step varies with the reaction rate. In general, the reaction time step is smaller than the diffusion time step, thus causing a workload imbalance. The calculation continues until the time rate of change of the burning reactants is less than a specified constant.

Table E-1 gives performance results for the test problem run on the MTA, V2250, and X2000. The single-processor speed on the MTA is 2.5 times faster than on the V2250 and 3.7 times faster than on the X2000.

Moreover, MTA scalability is much better, indicating very effective load balancing. At 8 processors the parallel efficiency is 92% on the MTA as compared to 43% and 47% on the V2250 and X2000, respectively. Load imbalance and system overhead for thread management account for much of the poorer scalability on the HP systems. Speedup peaks at 16 processors on the V2250 and at 12 processors on the X2000.

Table E-1. Performance results for Explode with 301x301 mesh points

Computer & f_s	Processors	Time (s)	Speedup	Measured efficiency	Fit efficiency
MTA	1	31.23	1.00	1.000	1.000
$f_s = 0.013$	2	15.95	1.96	0.979	0.987
	3	10.67	2.93	0.976	0.975
	4	8.07	3.87	0.968	0.963
	5	6.55	4.77	0.954	0.951
	6	5.57	5.61	0.935	0.939
	7	4.82	6.48	0.926	0.928
	8	4.24	7.36	0.921	0.917
V2250	1	78.0	1.00	1.000	1.000
$f_s = 0.20$	2	47.8	1.63	0.816	0.833
	4	31.8	2.45	0.613	0.625
	8	22.8	3.42	0.428	0.417
	12	18.2	4.29	0.357	
	16	12.7	6.14	0.384	
	20	17.5	4.46	0.223	
	32	20.7	3.77	0.118	
X2000	1	116.3	1.00	1.000	1.000
$f_s = 0.15$	2	65.8	1.77	0.884	0.870
	4	42.2	2.76	0.689	0.690
	8	30.7	3.79	0.474	0.488
	12	24.9	4.67	0.389	0.377
	16	30.8	3.78	0.236	

Performance on the HP systems would likely improve if the algorithm were written in a more coarse-grained manner, while paying close attention to data distribution in memory and between threads. However, the associated programming effort would be non-trivial and substantially greater than on the MTA, which achieves efficient multithreading without regard to data distribution.

An interesting finding is that scalability on the MTA is fit very well by Amdahl's law:

$$\text{efficiency} = 1 \,/\, (1 + f_s(p-1)),$$

with a serial fraction f_s of 0.013, where p is the number of processors. The efficiency measured and that fit by Amdahl's law are listed to three figures in the table. For all numbers of processors the agreement is good to less than 1%. This also demonstrates no significant load imbalance.

Amdahl's law also fits to within 2% the V2250 and X2000 efficiencies up to 8 and 12 processors, respectively, but for much larger values of f_s, as noted in the table. These larger effective serial fractions include the effects of poor load balancing and system overhead.

Conclusion

The thermal explosion benchmark representing highly dynamic workloads performed very well on the MTA. No special attention was given to data locality when implementing the multithreaded version, which made coding straightforward. Excellent scaling was obtained, which bodes well for other codes that require good load balancing to manage variable workloads.

Reference

E-1. M. Short and J.J. Quirk, "On the Nonlinear Stability and Detonability Limit of a Detonation Wave for a Model Three-Step Chain-Branching Reaction," *Journal of Fluid Mechanics*, 339, 89-112 (1997).

Synthetic Aperture Radar

Timothy P. Boggess and Stephen R. Blatt
Sanders/Lockheed Martin

Summary

Sanders, a Lockheed Martin Company, has ported the RASSP SAR benchmark [1] to the Tera MTA computer. This report documents the result of that effort and explores the viability of the MTA for real-time embedded applications such as signal processing.

Sanders is building some of the most powerful embedded multiprocessor systems in the world, interconnecting hundreds of processors. Multithreaded architectures provide an innovative way to get more performance out of a individual processor by ensuring that it is never idle, whether due to a cache miss, the inherent speed of the memory, or data synchronization. In addition, multi-threading promises much easier code parallelization.

Sanders' goal in this program was to evaluate a particular multithreaded architecture, the Tera MTA, for real-time, embedded applications to determine

- what performance increases might be expected over conventional processing,
- how the hardware and software scale as the number of processors increases, and
- how the latency might be controlled so that task deadlines are not missed.

The investigation was broken into two tasks: 1) evaluation of current MTA systems for real-time embedded applications, and 2) a study to identify what architectural changes would be appropriate for a variant MTA specifically designed for real-time embedded applications.

The evaluation included performance metrics, such as processor utilization, scalability, and real-time capabilities. Also considered were usability metrics, such as ease of programming, tools for tuning, and software portability.

The principal findings are that:

- The MTA is relatively easy to program.
- Tera provides good tools for isolating and tuning unoptimized code.
- Real-time performance is achievable on the chosen benchmark with only 3 MTA processors, while 8 processors beat the requirements for real-time operation by more than a factor of two.
- The MTA is a good soft real-time computer.
- Improvements in communications, task scheduling, form factor, and fault tolerance would be needed for the MTA to become a hard real-time computer.

Objective

The objective of this study was to determine the suitability of multithreading as implemented by Tera Computer in its MTA computer for use in high-performance, real-time applications developed by Sanders. The particular application investigated was Synthetic Aperture Radar (SAR) signal processing. Specific elements evaluated included:

1. System performance (speed and scalability) with a known benchmark,
2. Ease of porting existing code to a new computer environment,
3. Identification of issues impeding real-time usage of the MTA.

Description of Application

The SAR application investigated was the RASSP Benchmark-1. This code was developed by the MIT Lincoln Laboratory for its Advanced Detection Technology Sensor (ADTS), which is a polarimetric air-to-ground SAR operating in strip-map mode.

Synthetic Aperture Radar is a technique to combine multiple radar returns coherently to achieve an effective increase in the aperture size. An example flight path for an air-based system is shown in Figure F-1. By combining returns along the flight path, the cross-range (or azimuthal) resolution is increased.

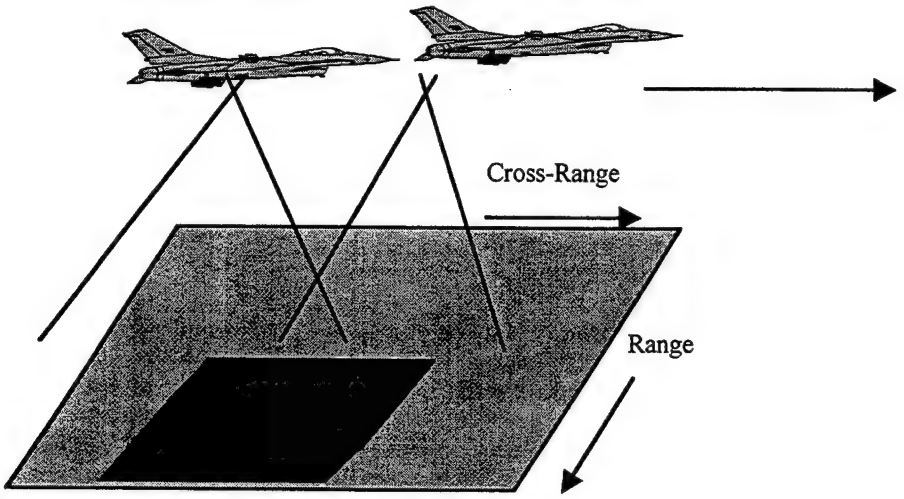


Figure F-1. Example SAR geometry

The stages for processing the SAR data are shown in Figure F-2. The data are initially read in a format specific to the Lincoln Laboratory ADTS system. Each data frame consists of 512 pulses that are acquired in 0.92 s. Each pulse provides a packet of data, preceded by a Barker code header. The algorithm identifies the start of data by finding the Barker code. Then range processing is done, which consists of applying Taylor weighting, performing a 2,048-point FFT, and applying radar cross-section weighting. This is followed by azimuth (or cross-range) processing, in which the data from all 512 pulses are combined and convolved with a range-based kernel. This involves performing a 1,024-point FFT, multiplying by the kernel, and performing the inverse 1,024-point FFT to provide the output. For validation purposes, checksums are calculated across one row and one column and compared to the pre-calculated checksums for the specific data sets.

The benchmark code is written in C. It consists of a main routine and supporting routines to read in data and perform FFTs. The software package also includes weights and coefficients required by the algorithm and software to generate test data sets.

Porting and Tuning

The porting and tuning of the SAR benchmark for the MTA consisted of identifying and changing variable declarations to accommodate bit-level manipulations in the original code, restructuring the way data were introduced into the processing, and adding compiler directives to help parallelization. The tuning tools Canal and Traceview were quite useful in determining what needed to be done. However, running Traceview over the Internet was extremely slow.

Much of the time and many of the problems encountered in porting and tuning were associated with compiler changes, operating system changes, and communications issues arising from the immature software environment of the MTA. The effort was helped considerably by assistance from Tera staff.

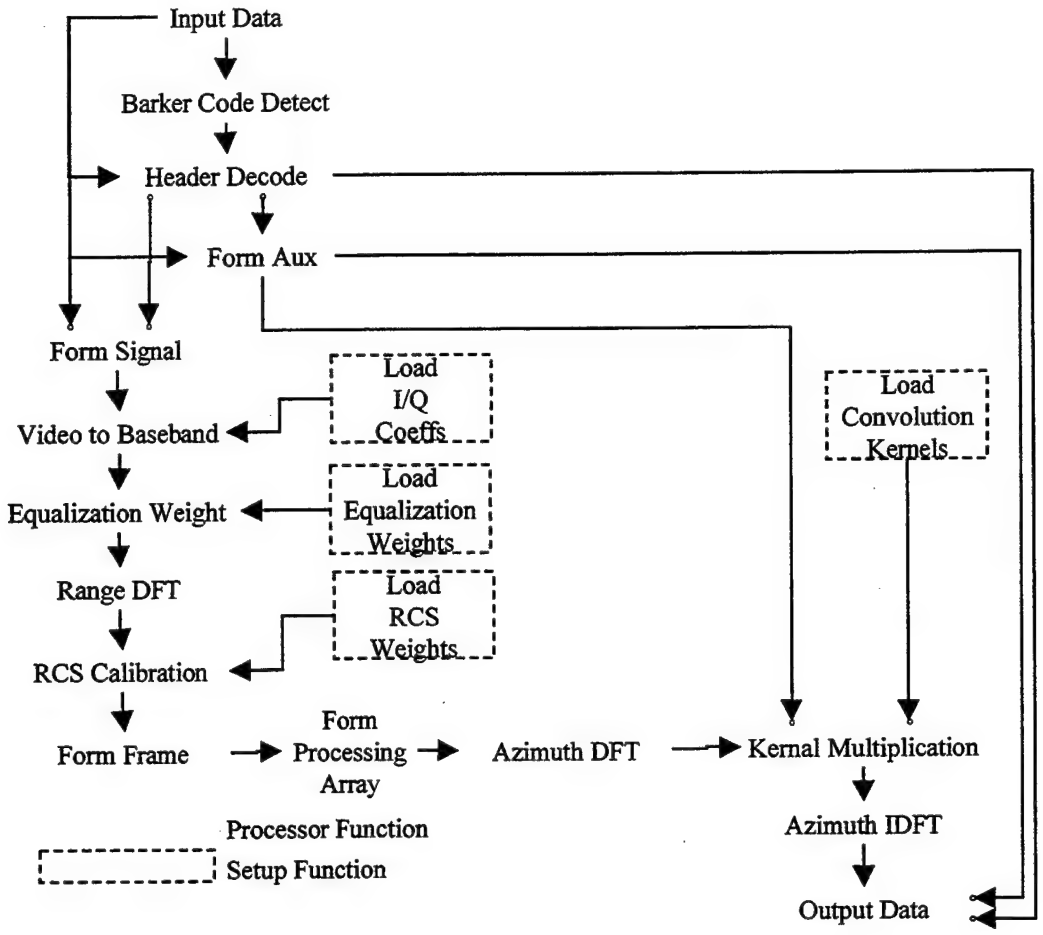


Figure F-2. SAR algorithm process flow

Porting and tuning involved several steps:

- Port original software to MTA.
- Add explicit parallelization directives.
- Revise data reading strategy.
- Revise algorithms.
- Remove tracing to obtain faster timing.

Further discussion of each of these follows.

Initial Port. The original software compiled, linked, and ran in serial mode on MTA, but gave incorrect results (as indicated by the checksum). This was traced to the way that data were read in and manipulated at the bit level. Since the MTA has a 64-bit processor, its default word size for integers is 8 bytes. This is twice as long as the 4-byte default on the 32-bit machines originally used to develop the benchmark. Once key variables were declared as integer*4, the benchmark executed correctly in serial mode on the MTA and processed one frame of data in 78 gigaticks (or 300 s for a 260-MHz clock). (Run times during for the porting and tuning are presented in gigaticks to eliminate variations associated with changes in the clock speed. One tick = one clock period = 3.85 ns for a 260-MHz clock.) For reference, the original benchmark took 32 s to process a single frame on a 200-MHz Sun UltraSPARC II workstation.

Parallelization. Compiler directives regarding parallelization were inserted in the main routine ahead of each major for loop to parallelize the azimuth processing. These directives and their purpose are as follows:

#pragma tera assert parallel: to assert that the separate iterations of a loop are independent and may execute concurrently.
#pragma tera dynamic schedule: to assign iterations to threads at run-time rather than assigning them by blocks at compile time. Dynamic assignment provides more flexibility if some iterations take longer than others.
#pragma tera use n streams: to request at least n threads are used per processor for a loop. At this stage the compiler default of 40 streams was used.

Implementing these directives and increasing the number of processors from one to two reduced the run time from the serial time of 78 gigaticks to 38 gigaticks.

Revised Data Input. The original code read in data from a file. However, in a real-time implementation, the actual data would be written to memory through Direct Memory Access, and the overhead of accessing disk would not be present. Furthermore, the original code read in each pulse (out of 512 within a frame), range-processed it, and went back to read the next one.

Two significant changes associated with data input were made. First the code was modified to read in the entire data file at the beginning of execution. This reduced the run time to 6.5 gigaticks. Then the time to read in the file was eliminated from the measured time, since data input would be handled by a separate processor in a real-time setting. This further reduced the run time to 1.3 gigaticks on two processors, showing that all of the preceding run times were dominated by data input.

Utilizing a Special Operation. The FFT algorithm was modified to replace a small routine that implemented a bit-reversal of indices by a special operation on the MTA: TERA_BIT_MAT_OR. This reduced the run time to 1.0 gigaticks.

Timing without Traceview. Execution with Traceview was useful in identifying implementation inefficiencies, but added overhead to the run times. Removing Traceview from the executable reduced the run time to 0.50 gigaticks.

Further Tuning. Increasing the number of streams per processor to 100 or more and the number of processors from two to four reduced the run time to 0.19 gigaticks. When the system size increased to eight processors at the end of the project, the run time further decreased to 0.11 gigaticks (or 0.43 s). For reference, the same code ran on the 200-MHz Sun workstation in 26 s.

Scaling Results

Timing results for one through eight processors, broken down by various components, are listed in Table F-1 and plotted in Figure F-3. The time to process one frame with eight processors is 18% of the time with one processor. Compared to the ideal of 12.5% of the single-processor time, this corresponds to 69% parallel efficiency. The time for azimuthal processing, the dominant component, decreases the most on eight processors to 16% of the single-processor time, while the times for range and header processing decreas to 22% of the single-processor times.

Table F-1. Times for single frame of SAR benchmark on MTA

Component	1 processor	2 processors	3 processors	4 processors	6 processors	8 processors
Header time (s)	0.09	0.05	0.04	0.03	0.03	0.02
Range time (s)	0.72	0.38	0.28	0.26	0.16	0.16
Azimuth time (s)	1.58	0.79	0.55	0.44	0.32	0.25
Total time (s)	2.38	1.22	0.86	0.73	0.51	0.43
Parallel efficiency	100%	98%	92%	82%	78%	69%

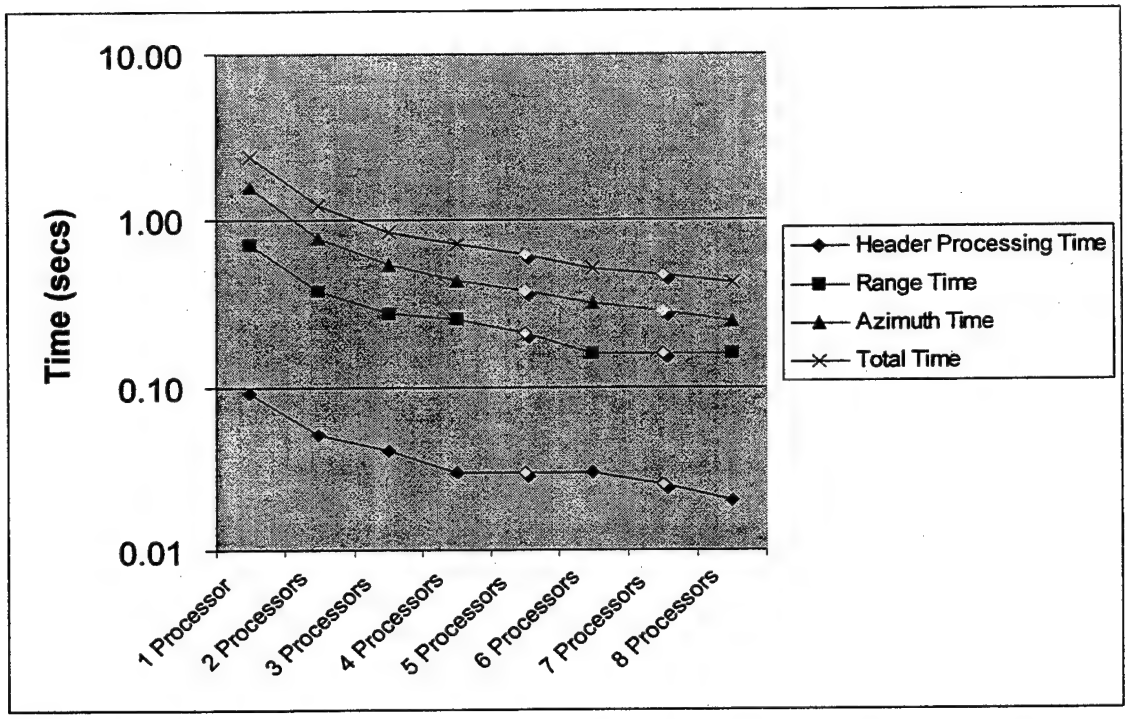


Figure F-3. Timing results for one frame of SAR data versus number of processors. (Note that shadowed points for 5 and 7 processors are interpolated.)

Even though the azimuthal component is dominant, scalability is primarily limited by the range component, the time for which does not improve at all going from 6 to 8 processors. Examination of the range processing code reveals only one significant loop, which has too few iterations to use more than a few processors effectively.

Table F-2 shows that the times for processing different data frames are constant as the number of frames increases.

Table F-2. Times for multiple frames of SAR benchmark on MTA

Component	1 Processor	2 Processors	3 Processors	4 Processors
Frame 1 time (s)	2.35	1.23	0.84	0.73
Frame 2 time (s)	2.35	1.24	0.84	0.73
Frame 3 time (s)	2.37	1.24	0.85	0.74
Frame 4 time (s)	2.41	1.27	0.87	0.74
Total time (s)	9.48	4.98	3.40	2.94
Parallel efficiency	100%	95%	93%	81%

Figures F-4 and F-5 give Traceview plots showing processor utilization for two different cases. The first plot corresponds to processing a single frame on four processors at an intermediate stage of tuning. The x-axis represents time, with the units being gigaticks. Three lines are plotted:

- The uppermost, medium-shaded line shows the number of instructions available each tick. Since there are four processors on a dedicated machine, the line approaches 4 instructions/tick for the entire run. A small fraction of the instructions is consumed by the operating system.
- The bright line shows the number of instructions per tick executed by the program. When compared to the preceding line, this gives the processor utilization. At the left of the plot, during initialization, the number is very small. By contrast, during the subsequent process-

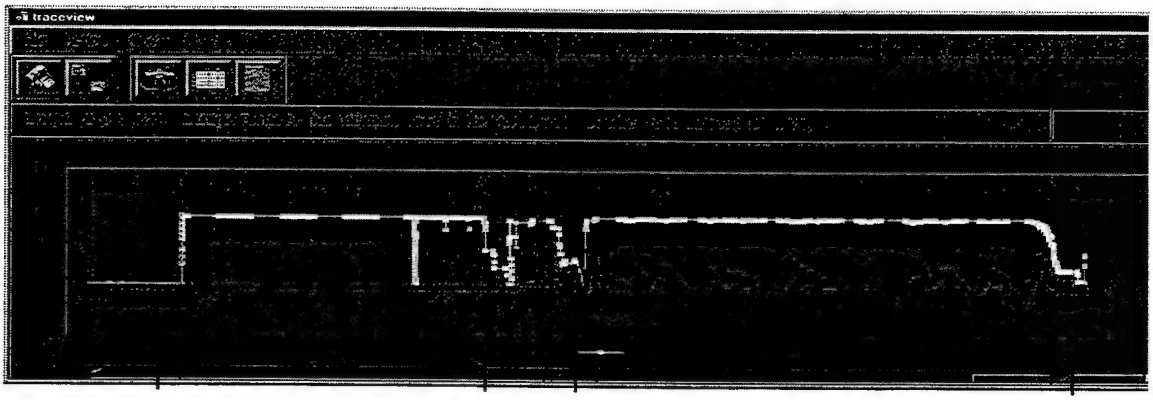


Figure F-4. Traceview output for one frame with four processors available.

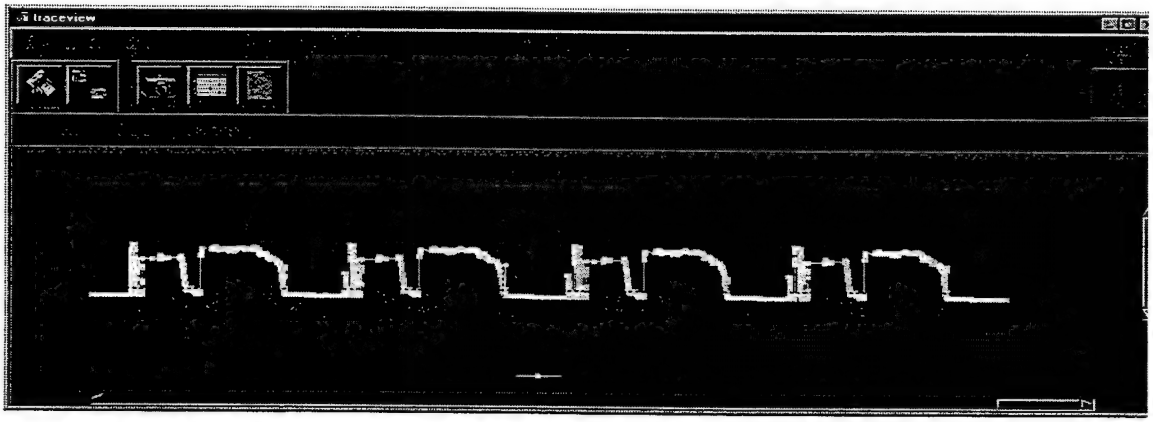


Figure F-5. Traceview output for four consecutive frames with eight processors, showing header, range, and azimuthal processing structure for each frame.

ing components, all four processors are fully utilized except at the beginning and end of each component. Note that header processing is relatively time-consuming, more than range processing in fact, at this stage of tuning.

- The dark line shows the number of floating-point operations per tick. Essentially all of the flops occur during the range and azimuthal processing components.

The second plot corresponds to processing four successive frames on eight processors at the end of tuning. This time the units for the x-axis are seconds, and the flop rate is not shown. The vertical scale is doubled, since there are twice as many processors, and the horizontal scale is compressed to accommodate four frames. Otherwise the structure is similar to that in the previous plot, except that the header processing time is relatively short now.

Real-Time Considerations

Real-time processing requires a run time less than the data acquisition time, which is 0.92 s per frame. Examination of the scaling data in the preceding tables shows that 3 MTA processors are sufficient to process the data in real-time, whereas 8 processors run more than twice as fast as necessary.

Systems developed for real-time applications are generally classified as soft real-time or hard real-time systems. A soft real-time system is designed so that timing deadlines are rarely missed, but without deadline assurances being built in. A soft system is suitable for applications where late execution is unfortunate, but not critical. Hard real-time systems, on the other hand, are built to

ensure timing performance and predictability. Tasks must have known durations (e.g., FOR loops rather than WHILE loops), and error handling must be built in to handle missed deadlines. Mission-critical applications are hard real-time. As examples, RealPlayer audio/video is soft real-time, whereas a missile defense system is hard real-time.

As a soft real-time computer, the MTA performs well. Its multithreaded architecture and near-linear scalability provide the capability for enough processing power to keep problems with soft time requirements (such as the SAR benchmark) running in real-time.

Key attributes of hard real-time systems are the following:

- A global clock is present to maintain timing across the system.
- Interrupts are time triggered rather than event triggered. When an exception occurs in an event-triggered system, a cascade of triggers often results, which overwhelms the operating system.
- Language support is available to identify and discourage hard-to-time constructs, such as WHILE, and to specify what to do when a deadline is missed (e.g., stop processing the frame that is now late and start on the next one, so that the system does not fall behind).
- Tasks are real-time scheduled and executed according to priority. In general, data rates are constant, and input signals are known, so tasks can be prioritized to assure completion of every task prior to its deadline.
- Communications use protocols that are time predictable, such as TTP (Time Triggered Protocol), rather than protocols with unbounded latency, such as Ethernet.
- Fault tolerance is generally required so that the system as a whole continues to function even if one of its components goes down.

Of these requirements, the only one currently available in the MTA is a global clock. Real-time scheduling, time triggers, and language support are not presently included, but could be added relatively easily. The major areas in which improvements would be needed are predictable real-time communications and fault tolerance.

The real-time communications issue arises because of the MTA's network design, which employs non-deterministic packet routing and makes no real-time guarantees. Network nodes use a form of deflection routing, so packets are not buffered. At each clock tick a network node reads the incoming packets from each port, chooses a destination port for each packet, and routes the outgoing packets. Since only one packet may be routed to a given port each tick, collisions are resolved by randomly assigning an unused port to all but one of the packets competing for a given port. Therefore, the route a packet travels through the network is non-deterministic, and no guarantee can be made about travel time. To ensure message delivery within a fixed time, the network would need to be redesigned.

Ironically the MTA is designed to tolerate large memory latencies. For a hard real-time system, however, an upper bound on the allowable memory latency is evidently needed.

One final consideration is that real-time systems are often embedded, so size, weight, and power (SWaP) are important. Although these factors do not affect real-time correctness, customers who need real-time predictability generally need to minimize SWaP as well. Since the MTA was designed to be a supercomputer, it is large, heavy, and very power-hungry. As a result, few real-time applications could use an MTA as is, even if it were hard real-time compliant.

Changes to the MTA design underway or under consideration could make the MTA more suitable for real-time applications. The processor and memory boards are being redone in CMOS parts rather than GaAs ones; the chip count is being reduced considerably; the form factor is being reduced to about the size of a VME chassis; and alternatives to water cooling are being explored.

Conclusions

Performance and Scaling. Real-time performance for the SAR benchmark was achieved on a three-processor MTA system. The MTA exhibits good scalability from one up to the eight processors available for evaluation. Parallel efficiency varied from 98% with 2 processors to 82% with 4 proc-

essors to 69% with 8 processors. With some additional effort, even better scaling might be achieved by overlapping the processing of data frames. In any case, at least a factor of two more work could be accommodated on 8 processors and still meet the real-time requirement.

Porting and Tuning. The changes made to the benchmark code were relatively minor. Much of the time and many of the problems in porting and tuning were due to the immature software environment on the MTA. The tuning tools were helpful, but assistance from Tera staff was invaluable.

Although the SAR algorithm was not implemented in parallel on a conventional signal processor for a true comparison, the effort required to do such an implementation is believed to be considerably higher than the port to the MTA. The reduced labor for algorithm porting to the MTA is one of its most attractive features.

Real-Time Issues. Performance of the MTA is good enough for soft real-time applications, where occasional instances of late execution are acceptable. For hard real-time applications, changes in communications and task scheduling are needed. Moreover, for the defense environments in which real-time processing is needed, improvements in the MTA form factor and fault tolerance are also needed.

Acknowledgments

Thanks are due to Rome Laboratory for providing Sanders access to the RASSP SAR code and to Tera Computer Company, especially Preston Briggs, for providing programming guidance and technical consulting.

Reference

F-1. B. W. Zuerndorfer et al., "RASSP Benchmark-1 Technical Description," MIT Lincoln Laboratory Project Report RASSP-1, (December 1994); also llex.ll.mit.edu/llrassp/documents.html.

TRANAIR – Computational Fluid Dynamics

L.G. Stern, Kristyn Maschhoff, Leonid Zaslavsky, Dominik Obrist, and Patricia Burgess
Tera Computer

G.1. Introduction

TRANAIR is a large-scale fluid dynamics package developed and used extensively by The Boeing Company for aircraft design. Developed and optimized over many years for Cray vector supercomputers, TRANAIR is representative of many defense legacy codes in its complexity and lack of portability.

The goals of this project were to evaluate the feasibility of porting a production-level defense application to the Tera MTA and to investigate how easily multilevel parallelism can be applied to an industrial problem. This evaluation involved the identification of several time-critical sections within TRANAIR. To demonstrate how multithreading on the MTA can be used to accelerate and scale these critical sections, new algorithmic approaches were implemented.

Another important aspect of this work was the feedback provided to MTA system and compiler software developers. MTA system stability improved steadily and dramatically during the evaluation period, due in part to stress-testing TRANAIR modules.

This report outlines the accomplishments made, the porting effort required, and the algorithmic modifications implemented to take advantage of the multithreaded architecture. In Section G.2 the basic structure of the TRANAIR package is outlined. Section G.3 discusses the overall porting effort and identifies the critical sections of the code and the levels of parallelization available.

Sections G.4 and Section G.5 discuss the motivation and development of an incomplete LU factorization (ILU) algorithm optimized for the MTA. The most computationally intensive part of TRANAIR lies in solving the nonlinear system of equations that result from the finite element approximation. Its efficiency relies on the efficiency of the parallel linear preconditioner, in this case incomplete LU factorization, used in the hierarchical, preconditioned Krylov-Newton iterations. The preconditioner is a critical component in obtaining TRANAIR performance.

Performance results for the factorization and triangular solves are presented in Section G.6. Timings for the ILU factorization applied to matrices generated from a Boeing 747 test case on an eight-processor MTA show a 4.8x speedup over a single-processor run and a 2.1x speedup over the original version run on a single T90 processor. Section G.7 summarizes the findings and discusses future work.

G.2. Structure of TRANAIR

TRANAIR is a package of computer programs for analyzing complex configurations in transonic flow with subsonic and supersonic freestreams. TRANAIR produces a numerical solution of the full potential equation subject to a set of general boundary conditions and can handle regions with different total pressures or temperatures. A locally refined rectangular grid generated automatically within the TRANAIR code is used for the discretization of the boundary value problem. The method is described in detail in Refs. [G-1] and [G-2].

Structurally, TRANAIR is a collection of Unix scripts and executables that are coordinated via job control files. The TRANAIR package consists of four main program modules:

- Input Processor (**fdinp** (3D) or **tdinp** (2D)),
- Flow Solver (**fdsol** or **tdsol**),
- Output Processor (**fdout** or **tdout**),
- Binary Converter to generate a graphics output file(**fdcic**).

The code is written primarily in Fortran 77, but includes a specialized I/O and memory management library written in C. Also included are C routines to implement Cray intrinsic functions.

Production runs of TRANAIR at Boeing are still limited to Cray vector supercomputers. The program has been historically difficult to port to other architectures. Huge memory requirements made necessary the use of out-of-core solvers for Cray computers. The embedded nature of such solvers complicates the direct parallelization of existing algorithms. In addition, the code calls many specialized Cray library routines.

G.3. Porting Efforts

Several modifications to the Boeing build structure were needed in order to port TRANAIR to the MTA. The complexity of the TRANAIR build process required the development of a specialized cross-compilation environment between Sun workstations and the MTA. To facilitate the debugging effort on the MTA, a Sun workstation version of TRANAIR was developed to serve as a reliable reference point. A collection of small NASA test cases, which exercise large portions of the 2-D and 3-D solvers, was provided to by Boeing and were used for testing purposes. Such tests were prerequisite to running more difficult problems.

The TRANAIR development model, particularly the use of Cray-style update modules, made it necessary to use standard object format libraries in the development. This made inlining and interprocedural optimization impossible and limited much of the potential performance improvement.

The code is organized into several archives which are linked together to generate the program modules. The porting effort has focused primarily on the flow solver module **fdsol/tdsol**, which is the computational engine of the code. The output processor and binary converter modules have not been evaluated at this time.

Discussions with Boeing scientists David Young and Craig Hilmes led to the identification of several areas in the code that admit parallelization. Throughout TRANAIR, there are abundant opportunities for parallelization in memory management. Data are typically accessed indirectly via a buffering method and are stored in blocks, which may be in- or out-of-core. Variables may also span several blocks. Since each block access is independent, the data-access loops could be made parallel. Work on the parallelization of data access is still in progress.

Within the flow solver there are multiple opportunities for parallelism. The flow solver consists of the following modules:

- **Initialization** starts the simulation and reads and/or generates initial data;
- **OPRDEF** defines and computes the finite element operators that make up the system of equations;
- **BLCKR** reorganizes the equations to bring them into an appropriate form for the solver;
- **SOLVER** solves the system of nonlinear equations by applying a Newton-GMRES method;
- **PSOLVR** is a design version of SOLVER that allows for multiple right-hand sides;
- **ADGREF** carries out adaptive grid refinement based on the convergence of the current solution.

The module OPRDEF defines and computes the finite-element operators and should achieve nearly perfect scaling on the MTA. Its structure lends itself well to high-level parallelism since the operators for individual finite elements can be computed independently. Unfortunately the style in which the code is written does not admit parallelization by the simple addition of compiler directives. Work continues on rewriting the structure of the operator definitions such that the computation is represented as a single parallel loop over all elements.

The module BLCKR reorganizes the set of equations and the unknowns so that they appear in a more convenient form for the solver, i.e., in blocked form. On the higher level this process involves many inherently serial algorithms. One can expect at most vector-level parallelism in these routines. It has not been determined whether it is better to parallelize the vector loops involving the data of a single block, or to parallelize over all of the blocks, or to do some combination of the two.

The SOLVER module solves the system of non-linear equations using a Newton-GMRES approach. PSOLVR is the "design" or "optimization" version of the SOLVER module and accommodates multiple right-hand sides. PSOLVR readily admits coarse-grain parallelization, since each right-hand side represents an independent linear system to be solved. As the parallelization of a single right-hand side is the more challenging problem and of more interest to Boeing scientists, efforts here concentrated on the SOLVER module. The B747I test contains a single right-hand side.

Most of the code modifications, while tedious and time-consuming, are theoretically not difficult to implement. The parallel sparse factorization of irregular systems present in the SOLVER module, however, has always been considered challenging in practice. Although the work on porting and tuning the full TRANAIR package continued throughout the project, given the limited availability of machine resources, the decision was made to focus on solver development early on as a means to demonstrate potential prior to a working implementation.

G.4. Motivation of Solver Development

From the experience of Boeing scientists and studies at Tera, it was evident that one area in which multithreading could have a significant impact on TRANAIR was in the preconditioning of the linearized problem within the GMRES solver phase. This includes both the generation of preconditioners and their application. This area is representative of the critical sections seen in many finite-element applications. The generation (incomplete LU factorization) and the application (solution of triangular systems) of the preconditioners required by the GMRES solver are critical to performance. On large problems, these two sections alone consume at least 80 percent of the serial execution time.

The nonlinear discrete system arising from the finite-element approximation in TRANAIR is solved using a preconditioned Krylov subspace method embedded in an inexact Newton Method. A three-level hierarchical iterative process is used:

- **an inexact Newton iteration process** is used to solve the boundary value problem;
- **a preconditioned Krylov subspace method** is used to obtain an approximate solution of the linear system arising in the Newton iterations;
- **an incomplete sparse LU factorization** is used to precondition the Krylov iterations.

While in theory TRANAIR contains abundant opportunities for parallelization, the Boeing data structures and programming style make it difficult to express this parallelism without modifying the code significantly. The ILU factorization, however, is viewed as a black box and hence is a prime candidate for replacement. Requirements for such a replacement include significant performance enhancement without degradation in solution accuracy and a demonstrated robustness covering a large class of problems. Although three potential levels of granularity in a parallel implementation of LU factorization were identified long ago (see, for example, [G-6]), the parallel sparse factorization of irregular systems has always been considered challenging in practice, since it requires the handling of indirect addressing and load balancing.

The Boeing implementation of the TRANAIR preconditioner is an incomplete LU factorization (ILU) with geometry-based multilevel nested dissection ordering. It is inherently serial and provides only inner-loop parallelism. In addition, the implementation of the incomplete factorization is not pure, in the sense that some updates are skipped in order to reduce communication between block updates. Although this results in some reduction in quality, it simplifies data access.

Many different parallelization techniques have been developed to provide coarse-grain parallelism for LU and ILU factorization on distributed- and shared-memory computers. Some examples are the parallel version of geometric nested dissection [G-7] and parallel graph partitioning [G-9] combined with the parallel maximum independent set approach [G-4], as in [G-8]. Multiple levels of parallelism must be exploited to utilize all processors because the structure of the L and U factors changes significantly over the course of the factorization due to fill-in. For sparse matrix factorization, this requires the use of both coarse- and fine-grained parallelism.

Parallel approaches to multilevel nested dissection ordering schemes based on parallel graph partitioning are also well studied [G-9]. Typically, such algorithms are geared toward limiting com-

munication costs while carefully maintaining load balance for distributed-memory architectures. Geometric techniques partition regular grids efficiently, but are inefficient if a very fine-grained partitioning of a multiply refined grid is required. Parallel multilevel graph partitioning is a complicated procedure that includes coarsening, coarse-level partitioning, uncoarsening, and refinement of the partitioning [G-9]. Partitioning is first used to distribute the work among several distributed-memory processors, and then the less-expensive Luby maximal independent set algorithm is applied to treat the separators (see [G-8], Section 4).

The ability of the MTA to tolerate memory latencies permits the exclusive use of the inexpensive Luby algorithm. Initial application of the expensive and complicated graph-partitioning phase is not necessary.

G.5. ILU Factorization and Triangular Solver Modules

When a matrix is factored, the total number of operations for a given pivot is equal to that pivot's row degree multiplied by its column degree. For sparse matrices, the number of operations for a given pivot does not provide sufficient parallelism. Therefore, a set of independent pivots must be chosen so that several pivot updates can be made concurrently.

To provide coarse-grain parallelism, a sequence of independent sets of pivots is constructed using a modification of the randomized technique proposed by Luby [G-4]. Luby's algorithm was chosen not only because it is highly parallel, but also because it tends to generate large independent sets in a few iterations. An iteration of the algorithm consists of two steps: first, a random set of candidates is constructed using an acceptance criterion for the nodes based on the number of connections; second, an independent set is extracted from these candidates. The application here requires a large, but not maximal, independent set. Only a few Luby iterations are made.

The coarse-grain parallelism provided by Luby's algorithm is then combined with parallelism at two other levels: pivots are factored concurrently, with each factor performing updates concurrently to rows below the pivot; and individual update or fill-in operations are performed in parallel.

This parallelism is sufficient to saturate a multithreaded multiprocessor at all stages of factorization: first, independent sets of pivots are large while the system is sparse; second, the number of updates per pivot increases as the matrix becomes denser [G-3].

An LU factorization is considered incomplete when thresholding is used to reduce fill-in. If a location in the factorization is already occupied by a non-zero element, the update is applied without checking the threshold. If the location is currently unfilled, the update is checked against a relative threshold, and the non-zero storage for that location is created only if the absolute value of the update exceeds the threshold. This strikes a balance between conserving memory and retaining good preconditioner quality. By modifying the threshold, total fill-in is usually constrained to between two and four times the original number of non-zero elements; however if matrix quality is poor, GMRES will require more iterations. Boeing's rule of thumb is that if GMRES does not converge in 30 to 60 iterations, the preconditioner (the Jacobian of the finite-element operator) will be recalculated and re-factorized.

As the row and column density increases and the factorizable population decreases, the Luby sets become smaller, and it becomes more difficult and more expensive to find sets of independent pivots. When the ratio of population to column density reaches a user-defined minimum value, the solution switches to a minimum degree algorithm. Because of thresholding, the Luby independent-set algorithm can be used much longer than would be possible in a full LU decomposition. Typically, 90 percent of the pivots can be applied before switching to minimum degree. In full LU, the switchover occurs much sooner.

Besides exploiting multilevel parallelism in an adaptive fashion, the approach adopted provides a mechanism for keeping fill-in low, while still maintaining a high-level of accuracy. The following mechanisms are used for fill-in control. First, the acceptance probability of a candidate at the random stage of the Luby algorithm is in inverse proportion to its degree. Second, a post-filtering

stage is applied to remove a small number of nodes with high degree if they were included in the independent set.

For initial testing of the factorization, a linked list storage format was used. Unfortunately, this format inhibited access to the finest level of parallelism: individual updates to row elements as a pivot column is eliminated. A consequence of thresholding is that row and column degrees do not increase as much during the course of factorization as they would in a full LU decomposition. This means that as the computation proceeds, the column density decreases while the number of independent sets is also decreasing. It is therefore necessary that the matrix be stored in a form that allows access to all three levels of parallelism.

Currently, a hash table to store non-zero elements is used in combination with block linked lists to store corresponding row and column indices. This technique allows direct access to matrix elements, and all three levels of parallelism can then be exploited. This approach is very efficient in achieving "fixed-size scalability" for factorization. Once the factorization is completed, the hash table can be deallocated.

For the solution of triangular systems, an efficient dataflow approach has been developed that allows two levels of parallelism to be exploited [G-3]. Coarse-grain concurrency can be achieved by processing in parallel a large number of independent pivots, while fine-grain concurrency can be achieved either through parallel sum reduction or by performing the updates in parallel. Using Tera's lightweight synchronization, one can avoid the barriers typically present in most parallel implementations of triangular solvers.

G.6. Performance

ILU Factorization. Several matrices were used to test and tune the Tera-developed ILU factorization. The first matrix, provided by Dr. L. Wigton of Boeing, has 9k rows and 133k non-zeros. This matrix is considered difficult to factor with good quality. Four larger matrices, up to 139k rows and 8.76M non-zeros, were generated using the Sun version of TRANAIR. These correspond to the second through fifth grids in the B747 inviscid test problem.

Table G-1 contains timings for the sparse ILU factorization on the MTA and T90. The times in bold correspond to those for which the MTA is faster than the T90 and show that 3 or 4 MTA processors are comparable in speed to one T90 processor for the larger matrices. Also, for these matrices, 8 MTA processors are roughly twice as fast as a T90 processor.

Scaling on the MTA improves as the problem size increases, at least up to the two largest matrices, which are comparable. Even for those matrices, though, the parallel efficiency falls off appreciably with additional processors (to 60 percent at 8 processors), because the last part of the factorization has relatively little parallel work.

For the B747I matrices, the inviscid problem formulation gives rise to initially denser matrices that do not require a high-quality factorization (thus not needing fill-in that could create more parallel work for latter-stage minimum degree factoring). MTA performance should improve on "harder" problems, in which quality requires the matrices to be larger and sparser (because of adaptive refinement) resulting in more parallel work in the early stages. Likewise, more fill-in due to finer thresholding will yield more parallel work in the later stages.

It is worth noting that ILU factorization is a difficult algorithm to implement on multiple processors, even though it is recognized as the best quality preconditioner for iterative solvers. Most commercial codes do not even attempt a multiprocessor implementation of ILU factorization.

Also, Boeing's TRANAIR programmers have been tuning the code on vector processors for at least 10 years. In that time they have not pursued a Cray multitasking strategy because of the difficulty of doing so. Therefore, the Boeing algorithm does not scale at all on the T90. By contrast, the MTA algorithm has been tuned for only several months, so there is still room for improvement.

Figure G-1 is a Traceview output from the B747I-3 test case on 8 MTA processors. It shows that there is already considerable parallelism in an early grid for the 747 test case, even if full processor

Table G-1. Performance results for ILU factorization on MTA and T90

Data set Rows / non-0s	Threshold: MTA / T90	Fill-in: MTA / T90	MTA 1p time (s)	MTA 2p time (s)	MTA 4p time (s)	MTA 8p time (s)	T90 1p time (s)
Wigton 9,106 / 133,206	1.E-4	4.07	6.16	5.09	4.37	4.19	
B747I-2 17,849 / 953,190	1.E-3 / 1.E-3	1.29 / 1.11	16.80	10.59	8.46	7.55	5.22
B747I-3 58,536 / 3,405,266	1.E-3 / 1.E-3	1.56 / 1.62	54.57	33.31	**22.32**	**17.03**	25.87
B747I-4 * 115,209 / 6,760,002	7.5E-4 / 1.E-3	1.78 / 1.86	131.19	69.58	**46.51**	**27.55**	58.52
B747I-5 * 138,589 / 8,759,372	5.0E-4 / 1.E-3	2.01 / 1.96	213.93	114.13	**66.33**	**44.55**	79.36

* The MTA threshold was dropped slightly to improve ILU quality and keep fill-in about the same as on the T90.

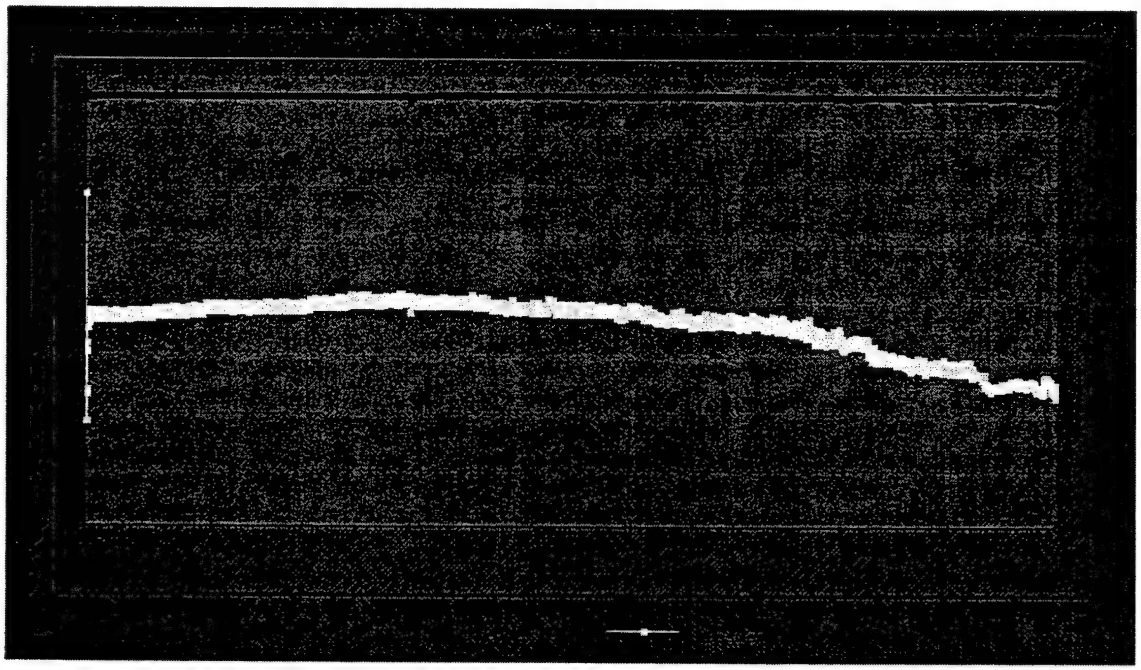


Figure G-1. Traceview output

utilization is not achieved. Furthermore, it indicates that improvements in the compiler would allow even better performance on this problem.

Flow Solver. The flow solver, which incorporates the ILU factorization, has been implemented on the MTA and used to run several of the NASA test cases (mentioned in Section G.3) to completion. The larger test cases have been hindered by time requirements for the un-parallelized linear solves.

Full integration of the Tera-developed ILU preconditioner should significantly speed up solver times and allow testing of larger problems.

Currently 70 percent of all routines in the flow solver module are compiled with compiler-generated parallelization enabled. Much of this parallelization, primarily loop-level parallelization, is guided by Cray directives already present in the code. Most of the efforts so far have been directed towards getting the NASA examples to run correctly to completion. With this almost complete, tuning to take better advantage of the parallelization opportunities discussed in Section G.5 will begin.

G.7. Conclusions

The TRANAIR port provided Tera a unique opportunity to test its architectural principles on a large, complex, fluid dynamics code considered challenging for most computer architectures.

Development of a new algorithmic approach for incomplete LU factorization demonstrates how multithreading on the MTA can be used to accelerate and scale a critical section in TRANAIR. Timings for the ILU factorization applied to matrices generated from a Boeing 747 test case show that 8 MTA processors are roughly twice as fast as a single T90 processor. While much work still needs to be done to improve the overall performance of TRANAIR on the MTA, several NASA test examples can be run to completion.

As a side effect of extracting parallelism in the ILU preconditioner, algorithms and data structures were developed that allow on-the-fly matrix reordering *based on data* as well as connectivity. This makes possible the development of robust, parallel partial pivoting strategies, which could lead to better quality preconditioning in a wider variety of industrial applications. Such pivoting strategies, impractical on other parallel architectures, are possible only because of the MTA's lightweight synchronization and much lower memory latency.

An important aspect of this work was the feedback provided to system software developers. Although this project was quite ambitious given the fragility of the MTA system at the beginning of the evaluation period and the known difficulties in porting this code to architectures other than Cray, persistence resulted in a software environment better able to handle the idiosyncrasies presented to it by legacy codes. In addition, the design and optimization of the parallel linear solver has helped to verify and improve the major paradigms of Tera compiler technology, in particular, the ability to exploit and schedule nested parallelism.

References

G-1. David P. Young, Robin G. Melvin, Michael B. Bieterman, Forrester T. Johnson, Satish S. Samant, and John E. Bussoletti, "A locally refined rectangular grid finite element method: Application to computational fluid dynamics and computational physics," *J. Comp. Phys.*, 92(1): 1-66 (1991).

G-2. F.T. Johnson, S.S. Samant, M.B. Bieterman, R.G. Melvin, D.P. Young, J.E. Bussoletti, and C.H. Hilmes, "TranAir: A Full-Potential, Solution-Adaptive, Rectangular Grid Code for Predicting Subsonic, Transonic, and Supersonic Flows About Arbitrary Configurations," NASA Contractor Report 4348 (December 1992).

G-3. Leonid Y. Zaslavsky, Simon H. Kahan, Bracy H. Elton, Kristyn J. Maschhoff, and Louis G. Stern, "A scalable approach for solving irregular sparse linear systems on the Tera MTA multithreaded parallel shared-memory computer," *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio TX (March 1999).

G-4. M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM J. Comput.*, 4:1036-1053 (1986).

G-5. Leonid Oliker and Rupak Biswas, "Efficient Parallelization of a Dynamic Unstructured Application on the Tera MTA,"Lawrence Berkeley National Laboratories Report #43190 (May 1999).

G-6. J. Liu, "Computational models and task scheduling for parallel sparse Cholesky factorization," *Parallel Computing*, 3: 327-342 (1986).

G-7. M.T. Heath and P. Raghavan, "A Cartesian parallel nested dissection algorithm," *SIAM. J. Matr. Anal. Appl.*, 16(1): 235--253 (1995).

G-8. George Karypis and Vipin Kumar, "Parallel Threshold-based ILU Factorization," Technical Report #96-061, Department of Computer Science, University of Minnesota & Army HPC Research Center (1996).

G-9. George Karypis and Vipin Kumar, "A Parallel Algorithm for Multilevel Graph partitioning and Sparse Matrix Ordering," *Journal of Parallel and Distr. Comput.*: 48: 71-95 (1998).

Electromagnetics

Joseph W. Manke, Subhankar Banerjee, Joel E. Hirsh, and Thomas W. Wicks
The Boeing Company

Overview

Boeing researchers evaluated the industrial usability, scalability, and performance of Tera's Multithreaded Architecture (MTA) computer for computationally intensive electromagnetics applications. Boeing's evaluation was oriented towards an in-depth look at today's applications, future requirements, and the processes that must be supported by the computing environment of the future.

The applications evaluated on the Tera MTA were Boeing's production-grade computational electromagnetics code called PARADYM and a kernel of the production code called the FMM Prototype. Based on the Method of Moments (MoM), the GMRES iterative method, and the multilevel Fast Multipole Method (FMM), PARADYM is representative of the best numerical algorithms used at Boeing for computationally intensive applications. Thus the evaluation of the performance and scalability of PARADYM on the MTA is applicable to the analysis of Boeing's future computing requirements for defense applications. Also, PARADYM currently runs on both shared- and distributed-memory platforms. Thus the performance and scalability results on the MTA can be compared to similar results on other architectures, such as the SGI Origin2000 (O2k) and clusters of PCs.

PARADYM and the FMM Prototype were ported to and tuned on the MTA. The tuning effort involved analysis and re-design of the implementation of the multilevel FMM to expose enough parallelism to exploit the multithreaded architecture. Once the tuning effort was completed, a scalable set of test cases was used to carry out performance and scalability studies.

This report describes the effort required to tune the codes to the MTA, presents the performance and scalabilty results, and gives an evaluation of the MTA with respect to of Boeing's future computing requirements for defense applications. A summary appears in the remainder of this section, while details are presented in subsequent sections.

Porting Codes to the MTA. Relatively little effort was required to port the FMM Prototype to the MTA. The problems encountered, which are typical of those in porting large applications to a new computer, involved dynamic memory allocation from Fortran, calling C code from Fortran, binary I/O, and differences in the specifications of the LOC and MALLOC functions between C and Fortran.

These problems were resolved for the FMM Prototype except for the largest test cases (with 250,000 or more unknowns). Dynamic memory allocation problems prevented multiprocessor execution of both the FMM Prototype for the largest test cases and the full PARADYM code for any of the test cases.

Tuning Evaluation Codes for Performance on the MTA. Tera's Fortran compiler, which supports both Tera compiler directives and Cray compiler directives for "DO-loop" level parallelism, made it easy to exploit coarse-grain parallelism in the evaluation codes. Moreover, Tera's Canal and Traceview tools were very useful in analyzing and tuning the evaluation codes for the MTA.

Some reprogramming of the FMM Prototype was necessary to achieve good performance and scalability of the multilevel FMM. The tuning effort involved collapsing several nested DO loops to obtain more threads.

Performance and Scalability of the MTA. Single-processor performance on the MTA was comparable to that on the O2k for the FMM Prototype on a test case with 250,000 unknowns. The trend in the data suggests that for full-vehicle radar cross-section simulations, where the number of un-

knowns is 1,000,000 to 10,000,000, single-processor performance on the MTA will be roughly twice that on the O2k.

The MTA does not scale the FMM Prototype as well as the O2k, however. For the test problem with 65,000 unknowns, the parallel efficiency at 8 processors is about 0.9 on the O2k, but only 0.7 on the MTA.

Industrial Usability of the MTA. Boeing requires systems that have not only a high level of performance, but also ones that are highly reliable and stable. The MTA installed at SDSC for this project was a development machine with rapidly evolving hardware and software. Although the resulting low availability and stability of the MTA made project work difficult, Tera did make great strides during the course of the project to improve MTA reliability and stability.

Method of Moments

The MoM algorithm is a frequency-domain technique for computing electromagnetic scattering from complex objects. The formulation of the MoM leads to a large, dense linear system of equations, which must be solved to compute the scattering. Traditionally, MoM algorithms have employed direct linear equation solvers for these systems. The high computational complexity of these direct MoM solvers has limited them to low-frequency problems. Recently, fast MoM solvers have been introduced with low computational complexity. These fast MoM solvers have the potential to solve larger problems at higher frequencies. Boeing's implementation of a fast MoM solver was used as the basis for evaluation of the MTA.

The scattering of a plane wave of a specified frequency from an object is given by Maxwell's equations. The Electric Field Integral Equation (EFIE) and the Magnetic Field Integral Equation (MFIE) formulations describe the surface current density induced by an incoming plane wave.

The MoM approach to solving the EFIE or the MFIE is to discretize the integral equation by expanding the current density in a set of N basis functions. Substituting this expansion into the integral equation, multiplying by a basis function, and integrating over the scattering surface reduces the problem to one of solving a dense linear system of equations,

$$Z\,j = v,$$

for the vector j of expansion coefficients. The entries in the matrix Z and the vector v are complicated double integrals over the scattering surface and must be calculated numerically.

The advantage of MoM algorithms is that they employ exact representations of Maxwell's equations, so highly accurate simulations are possible. The disadvantage of traditional MoM algorithms is that they are computationally intensive, especially as the frequency goes up. Generally N increases as the square of the frequency, where N is on the order of 1,000,000 for full-vehicle simulations.

In direct MoM solvers, which first appeared in the late 1960s, the linear system $Z\,j = v$ is solved directly. The computational complexity of direct MoM solvers includes $O(N^2)$ integral evaluations to compute the matrix Z and $O(N^3)$ arithmetic operations to solve the system $Z\,j = v$ for j. The memory requirement for direct MoM solvers is $O(N^2)$. For these reasons the direct MoM solvers are generally used only for low-frequency problems. Although direct MoM solvers have been highly optimized on various high-performance computers, including ones with shared and distributed memory, the largest problems solved so far are for N on the order of 100,000.

Recently, fast MoM solvers based on fast, iterative linear equation solvers have been introduced. The iterative solvers rely on numerically stable and rapidly converging iteration procedures, such as the preconditioned GMRES method [H-1]. Fast matrix-vector multiply algorithms are used to compute products of the form $Z\,x$ used in the iterative procedure. The computational complexity of the fast MoM solvers is $O(N\log N)$. The memory requirement for the fast MoM algorithms is $O(N)$. This is a remarkable reduction from the $O(N^3)$ computational complexity of the direct MoM solvers and allows the solution of much larger problems at higher frequencies.

Rohklin [H-2, H-3] introduced fast MoM solvers for the Helmholtz equation, which use iterative linear equation solvers and the fast multipole method (FMM) for fast matrix-vector multiplies. To compute products of the form Z x, the Z matrix is not formed or stored. Rather the product Z x is viewed as a field and approximately evaluated by the FMM. The mathematical formulation of the FMM is based on the theory of multipole expansions and involves translation (change of center) of multipole expansions and spherical harmonic filtering.

Building on the FMM approach, Dembart, Epton, and Yip [H-4 to H-7] of Boeing implemented a fast MoM solver in the production-grade electromagnetics code PARADYM used by the company for radar cross-section studies. Problems for which the number of unknowns is on the order of 10,000,000 have been solved with this code. Boeing's fast MoM solver uses the preconditioned GMRES iterative method, which requires only the calculation of products of the form Z x, combined with a multilevel FMM for fast matrix-vector multiplies.

Evaluation Codes

Two Boeing codes were used to evaluate the MTA for calculating electromagnetic scattering from complex objects by the Method of Moments.

The first code is PARADYM, which is structured as shown in Table H-1. The code is composed of four programs – G, F, S, and R – along with four supporting libraries – MSM, SINEX, ZFT, and UTIL. The G, F, S, and R programs must be run in sequence to compute the scattering from an object. The S program is the fast MoM solver, which uses the preconditioned GMRES iterative method and the multilevel FMM for fast matrix-vector multiplies. As indicated in Table H-1, both the G and the S programs and the UTIL library include Cray compiler directives for "DO-loop" level parallelism.

Table H-1. Structure of PARADYM code

Module	Function	Parallel Implementation
G	Generation of cube hierarchy and matrices (including preconditioner)	Parallel DO loops in three subroutines, Cray directives
F	Matrix factorization	None
S	Solution of dense linear system by preconditioned GMRES iteration using the Fast Multipole Method for matrix-vector multiplication	Parallel DO loops in one subroutine, Cray Directives
P	Improved translation and filtering loops	Parallel DO loops in three subroutines, Cray directives, performance tuning of translation and filter loops
R	Radar Cross-Section computation	None
ZFT	FFTs	None
UTIL	Dynamic memory management, cube hierarchy management, translation operators	Parallel DO loops in one subroutine, Cray directives
SINEX	Surface grid operations	None
MSM	More memory management	None

In previous Boeing work the P program was introduced. It replaces the S program and includes improved versions of the translation and filter loops (described shortly) from the FMM Prototype. As in the S program, the translation and filter loops include Cray compiler directives for "DO-loop" level parallelism.

The second evaluation code is the kernel of the fast MoM solver in the PARADYM code. Called the FMM Prototype code, it computes the "far-field" due to a distribution of scalar sources by the multilevel FMM. The FMM Prototype consists of 29,000 lines of Fortran code. The "translation" loops in subroutine EVL_TRNS and the "filter" loop in subroutine FMM_CFI are the key loops for coarse-grain parallelism in the FMM. These subroutines have exact counterparts in the P program of PARADYM. The translation and filter loops include Cray compiler directives for "DO-loop" level parallelism. Performance results and tuning information obtained from the FMM Prototype code can be directly transferred to PARADYM. The FMM Prototype has been extensively used at Boeing to develop parallel implementations of the FMM for shared- and distributed-memory platforms such as the SGI O2k and clusters of PCs.

Evaluation of Tera MTA

The first step was to establish a benchmarking environment for the evaluation codes and create a set of scalable test problems for performance and scalability studies. The evaluation codes were then ported to the MTA and tuned to the machine's multithreaded architecture. Finally, performance and scalability studies were carried out to evaluate the performance of the evaluation codes on the MTA. Also evaluated was the effort it took to tune the codes to the MTA to obtain good performance. Details on the evaluation process and the results obtained follow.

Benchmark Environment. To facilitate the work a benchmark environment was established at Boeing for the evaluation codes. The benchmark environment provided a set of scripts and make files to maintain the source code, test data, and test results; to compile the source code; and to run the test cases. Tera Computer Company provided Boeing the Fortran and C cross compilers for the MTA and the Canal and Traceview development tools for use in the benchmark environment.

Throughout the project, the PARADYM code was secured in the benchmark environment at Boeing. The Fortran and C cross compilers were used to build MTA executables for PARADYM at Boeing. Secure communications (SCP) were then used to transfer the executables to SDSC for testing on the MTA. This procedure was followed to meet the proprietary and security requirements specified by Boeing and Boeing's customers.

For the FMM Prototype only, a similar benchmark environment was also established at SDSC. This provided a convenient environment for working with the FMM Prototype at SDSC.

Benchmark Test Cases. A scalable set of benchmark test cases was developed to conduct performance and scalability studies on the MTA. The "flat-plate" series for PARADYM and the FMM prototype were designed to increase geometrically the number of unknowns in the dense linear system of equations presented to the FMM solver in PARADYM.

The flat-plate test cases for the FMM prototype are summarized in Table H-2. The frequency, wavelength, and cube hierarchy are specified for each test case. In the rows for each level, the first number in the column for a test case specifies the number of cubes for the level, and the second number specifies the order of the multipole expansions for the level. The table also shows the number of sources (which is approximately the number of unknowns in PARADYM) and the memory for signature function storage.

Porting Evaluation Codes to Tera MTA. The FMM Prototype code was first ported to the MTA. Relatively little effort was needed for this, though two problems were identified with dynamic memory allocation and inlining a logic function. Once these problems were resolved, the code was verified to produce correct results on the MTA. (The two compiler problems mentioned above were corrected in later versions of the compiler.)

Tera compiler directives were then used to achieve "DO-loop" parallelism in the translation and filter loops. Correctness of the parallel code when compiled with the –par option was verified for the 2x2, 4x4, 8x8, and 16x16 test cases. When the compiler was updated to handle Cray compiler directives properly, the Tera directives were replaced by Cray directives, and correctness was again verified. This allowed PARADYM, which uses Cray compiler directives for "DO-loop" parallelism, to be ported to the MTA with minimal changes. It also meant that performance and scalability data for the FMM Prototype could be used to predict the performance of the P program in PARADYM.

Table H-2. Flat-plate test cases for FMM prototype

Test case	2x2	4x4	8x8	16x16	32x32	64x64
Frequency (GHz)	0.6	1.2	2.4	4.8	9.6	19.2
Wavelength (m)	0.5	0.25	0.125	0.0625	0.03125	0.015625
Level-6						16 256
Level-5					16 128	49 128
Level-4				16 64	49 64	196 64
Level-3			16 36	49 36	196 36	729 36
Level-2		16 20	49 20	196 20	729 20	2916 20
Level-1	16 12	49 12	196 12	729 12	2916 12	11449 12
Level-0	49 6	196 6	729 6	2916 6	11449 6	40000 6
Sources	1,024	4,096	16,384	65,536	262,144	1,048,576
Memory (MB)	3.2	11.2	42.8	162	646	

All of the performance data reported here are for the version of the FMM Prototype that uses Cray directives.

Once the port of the FMM Prototype to the MTA was complete, attention shifted to porting the full PARADYM code to the MTA. Some preliminary modifications to the G and S programs and the UTIL library were made to coordinate with the benchmark environment. Also, the translation and filter loops from the FMM Prototype, which use the Cray compiler directives for "DO-loop" parallelism, were added to the P module. The modified code was verified to produce correct results on an SGI O2k at Boeing. Working in the benchmark environment at Boeing, the G, F, S, R, and P modules as well as the MSM, SINEX, ZFT, and UTIL libraries were then verified to compile correctly with Tera's Fortran compiler.

After this preliminary work, testing of the PARADYM executable began on the MTA at SDSC following the plan shown in Table H-3.

Table H-3. Test plan for porting and evaluating PARADYM code

Test	Plan	Objective	Status
1	Test G, F, S, R modules and MSM, SINEX, ZFT, UTIL libraries for -serial	Validate correct execution of PARADYM on MTA	Validation complete
2	Test G,S modules and UTIL library with Cray directives for -par	Evaluate parallel performance on MTA with no changes to code	Validation complete; performance study incomplete
3	Test P module with Cray directives for -par	Evaluate parallel performance on MTA using *basic* translation and filtering loops from FMM Prototype	Validation complete; performance study incomplete
4	Test P module with performance tuning from FMM Prototype	Evaluate parallel performance on MTA using *tuned* translation and filtering loops from FMM Prototype	Validation and performance study not done

Work began on Test 1 first, in which the code was compiled with the –serial option and checked for correct execution. Several porting problems were identified, involving dynamic memory allocation from Fortran, calling C code from Fortran, binary I/O, and differences in the specifications of the LOC and MALLOC functions between C and Fortran. These problems were resolved, and correct serial execution was validated on the MTA, resulting in successful completion of Test 1.

Work then proceeded to Test 2, where the G and S programs and the UTIL library were compiled with the –par option and checked for correct execution. The significance of this test is that the G and S programs and the UTIL library include Cray directives for "DO-loop" parallelism. Correct execution of this multithreaded version of PARADYM was validated on one MTA processor. This is an important result because it shows that the shared-memory parallel version of PARADYM runs on the MTA with only the few porting changes mentioned previously. However, due to problems with dynamic memory, correct execution was not achieved on multiple MTA processors. As a result, the performance and scalability studies planned for Test 2 could not be conducted.

Next Test 3 was conducted, in which the P module was compiled with the –par option and checked for correct execution. The importance of this test is that it uses the parallel translation and filter loops from the FMM Prototype, which include Cray directives for "DO-loop" parallelism. Again, correct execution of this multithreaded version of PARADYM was achieved on one MTA processor. As before, however, problems with dynamic memory prevented correct execution on multiple MTA processors. As a result, the performance and scalability studies planned for Test 3 could not be conducted.

Insufficient time remained to conduct Test 4, in which the parallel translation and filter loops in the P module were to be updated to reflect the tuning results from the FMM Prototype.

Tuning Evaluation Codes for Tera MTA. As mentioned previously, Cray compiler directives were used to achieve "DO-loop" parallelism in the translation and filter loops. The performance and scalability of the FMM Prototype with this approach to parallelism were found to be limited, evidently because of variation in the granularity of the multilevel FMM over the levels of the cube hierarchy.

In particular, the number of cubes at each level *decreases* geometrically from the finest to the coarsest level, whereas the order of the multipole expansions *increases* geometrically from the finest to the coarsest level. As a result, the amount of computational work is roughly constant over the levels, but the amount of parallelism, exploited in the parallel DO loops over cubes, decreases from the finest to the coarsest level. Consequently, the performance and scalability of the approach are limited on the MTA, as verified by Tera's Traceview tool. This is especially true for the filter loop where FFTs are invoked from within the loop.

Tera's Canal and Traceview tools were used to identify additional parallel opportunities beyond the "DO-loop" parallelism in the translation and filter loops. Three approaches were considered for tuning the translation and filter loops: two taking advantage of fine-grain parallelism within the loops and the third using "futures" to pipeline the FMM calculation for additional parallelism.

In the first approach, several of the subroutines called from within the loops were compiled with parallel compiler directives. For the translation loops, -par parallelism was applied to the pointwise multiplication of signature functions by translation operators. For the filter loops, -par1 parallelism was applied to the FFTs used in the filtering operations. Since the translation and filter loops are compiled with the –par compiler directive to invoke "DO-loop" parallelism, this strategy amounted to placing parallel regions within parallel regions. This failed to generate additional parallel performance. In fact, the overhead of initiating a parallel region from within a parallel region actually degraded performance.

The second approach was applied to the filter loop. The code loops over all cubes at a given level in the cube hierarchy and performs a sequence of "multiple" FFTs, on columns and rows of each cube, to accomplish the filter operation. For a multiple FFT on columns, the FFT code treats the cube data as row-wise vectors, and the transform operations are composed as loops over the row-wise vectors. The multiple FFT on row is similar. The code was modified to loop over all cubes and all columns (or rows) of the cube and perform "single" FFTs one column (or row) at a time.

The length of the modified loop is the product of the number of cubes by the number of columns (or rows). For a given number of levels in the cube hierarchy, the length of the modified loop is approximately the same over all levels in the multilevel FMM. For problems with four levels (16,000 or more unknowns), the length of the modified loop is large enough to obtain high parallel performance from "DO-loop" level parallelism. The parallel performance of the modified loop also showed good scalability.

The third approach used an MTA-specific feature called futures to pipeline the FMM calculation for additional parallelism. Futures allow all the translation and filter loops to run in parallel, while the loops themselves execute as parallel DO loops. The futures are synchronized by using the full/empty bit of the MTA. Data for an individual cube are used (consumed) by a translation or filter loop as soon as they are computed (produced) by other loops. Synchronization by testing the full/empty bit is a lightweight process on the MTA, so this producer/consumer approach should be efficient and scalable.

As it turned out, this approach did not lead to improved parallel performance on the MTA. The reason is that the MTA operating system assigns streams to futures in a static manner. When all parallel translation and filter loops were initiated at once using futures, the system could not assign enough streams to all of the loops. Some loops ran with very few streams and so ran very slowly. Indeed, the futures version of the FMM prototype ran slower than the standard version. Although this result is disappointing, it is possible that a more sophisticated strategy for initiating the loops might lead to improved parallel performance. Also, a future version of the MTA operating system is expected to support dynamic allocation of streams to futures.

Performance and Scalability Results. Throughout the project, the flat-plate test cases were used to measure the performance and scalability of the FMM Prototype on the MTA. These measurements, along with Tera's Canal and Traceview tools, also guided the tuning efforts. The lack of stability of the MTA made it very difficult to obtain reliable and reproducible performance data. The performance and scalability data presented here represent the best results obtained.

Table H-4 compares performance data for the FMM Prototype on the Tera MTA (at 260 MHz) and the SGI O2k (at 250 MHz). The numbers for each test case are wall-clock times in seconds measured during dedicated time slots on each machine. The data for the O2k were obtained with an MPI parallel version of the FMM Prototype. The data for the MTA were obtained with the tuned version of the FMM Prototype using the second approach described previously. Due to problems with dynamic memory, multiprocessor runs were not completed on the MTA for the 32x32 test case.

Table H-4. Performance data for FMM Prototype on Tera MTA and SGI O2k

N	2x2*		4x4*		8x8*		16x16*		32x32*	
	O2k	MTA	O2k	MTA	O2k	MTA	O2k	MTA	O2k	MTA
1	.262	.643	1.56	3.01	9.07	14.4	53.3	64.3	404.	383.
2	.174	.502	.826	1.65	4.68	7.16	26.7	35.0	189.	
4	.133	.475	.464	1.09	2.34	4.03	13.7	19.2	104.	
8	.220	.468	.307	.828	1.25	2.71	7.34	11.6	49.1	
16	.205		.307		.918		4.17		30.5	

* The tabulated numbers are wall-clock times in seconds.

Single-processor times on each machine are plotted versus the problem size in Figure H-1. The plot and the tabulated data show that the single-processor performance of the MTA just exceeds that of the O2k for the 32x32 test case with 250,000 unknowns. The trend in the data suggests that for full-vehicle simulations, where the number of unknowns is 1,000,000 to 10,000,000 (corresponding to test cases of 64x64 and larger), the single-processor speed of the MTA will be approximately twice that of the O2k.

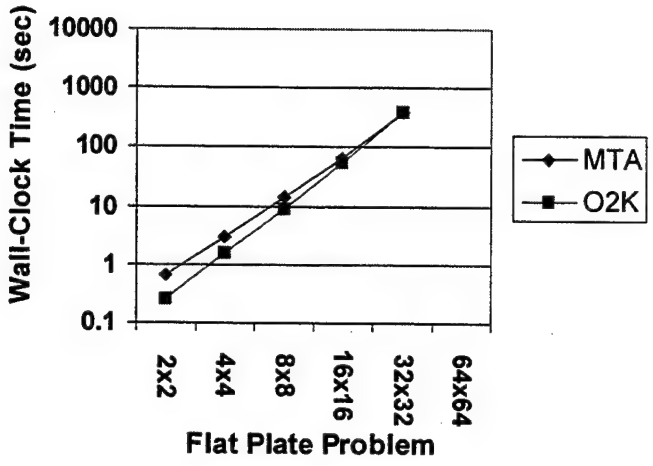


Figure H-1. Single-processor performance of FMM Prototype on Tera MTA and SGI O2k

Figures H-2 and H-3 show scalability on the MTA and O2k, respectively, for the various test cases. Scalability is appreciably better on the O2k. This is best seen by a direct comparison between the two machines, such as shown in Figure H-4 for the 16x16 test case, the largest for which there are data on both the MTA and O2k. Plotted there is the parallel efficiency (speedup divided by the number of processors) versus the number of processors (where ideal, linear scalability corresponds

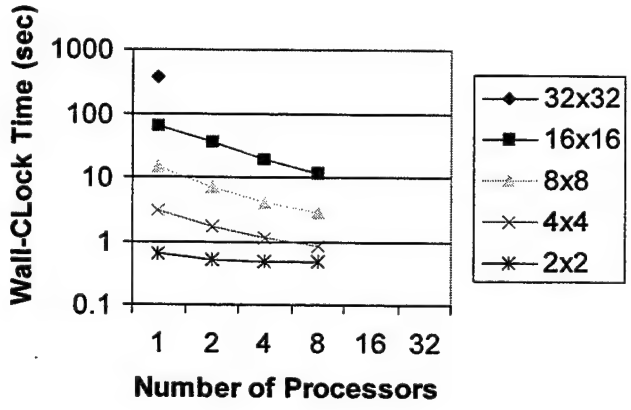


Figure H-2. Performance data for FMM Prototype on Tera MTA (260 MHz)

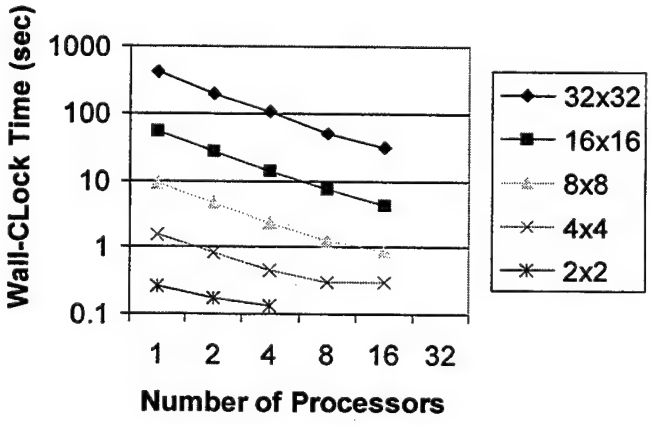


Figure H-3. Performance data for FMM Prototype on SGI O2k (250 MHz)

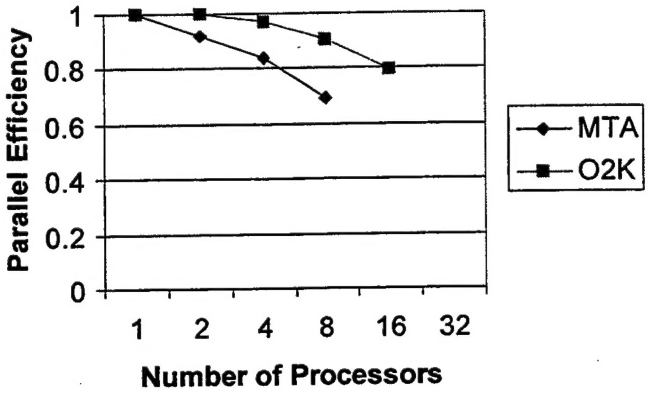


Figure H-4. Parallel efficiency for 16x16 test case on Tera MTA and SGI O2k

to a parallel efficiency of 1.0). The figure shows that at 8 processors, the parallel efficiency of the O2k is about 0.9, while the parallel efficiency of the MTA has fallen to about 0.7.

References

H-1. Yousef Saad, "Iterative Methods for Sparse Linear Systems," PWS Publishing Company, Boston MA (1996).

H-2. V. Rokhlin, "Diagonal Forms of Translation Operators for the Helmholtz Equation in Three Dimensions," Research Report YALEU/DCS/RR-894, Dept. of Comp. Sci., Yale Univ. (March 1992).

H-3. R. Coifman, V. Rokhlin and S. Wandzura, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription," *IEEE Antennas and Propagation Magazine,* 35(3), 7-12 (June 1993).

H-4. B. Dembart and E. L. Yip, "A 3-D Fast Multipole Method for Electromagnetics with Multiple Levels," ISSTECH-97-004, The Boeing Company (December 1994).

H-5. M. A. Epton. and B. Dembart, "Multipole Translation Theory for the 3-D Laplace and Helmholtz Equations," *SIAM J. Sci. Comput.* 16(4), 865-897 (July 1995).

H-6. M. A. Epton and B. Dembart, "Low Frequency Multipole Translation Theory for the Helmholtz Equation," SSGTECH-98-013, The Boeing Company (August 1998).

H-7. M. A. Epton and B. Dembart, "Spherical Harmonic Analysis and Syntheses for the Fast Multipole Method," SSGTECH-98-014, The Boeing Company (August 1998).

Symbiosis

Allan Snavely, Nick Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen
San Diego Supercomputer Center &
Computer Science and Engineering Department
University of California, San Diego

Symbiosis is defined as the mutually beneficial living together of two dissimilar organisms in close proximity. The term has been adopted to refer to the increase in throughput that can occur when two or more jobs are executed concurrently on a multithreaded computer. Work reported here has led to

- a formal definition of symbiosis in the context of multithreaded architectures,
- a series of empirical observations of the symbiotic effect on the Tera MTA, and
- development of a symbiotic job scheduler that increases throughput on a multithreaded computer.

This work is summarized here and described in more detail in Ref [I-1].

Formal Definitions

Consider a set of J jobs, each of which takes time t_j to execute by itself. Then the normalized throughput of running the jobs simultaneously instead of sequentially (i.e., one after another) is defined to be

$$TR = \text{sequential time} / \text{simultaneous time}$$
$$= (\text{sum } t_j \text{ for } j=1,J) / \text{simultaneous time.}$$

If TR is greater than 1, then there is an advantage to running the jobs simultaneously rather than sequentially. If TR is less than 1, the jobs interfere with each other, and it is better to wait for one to complete before running the next.

The best possible value for the normalized throughput is

$$TM = (\text{sum } t_j \text{ for } j=1,J) / (\max t_j \text{ for } j=1,J).$$

This could be obtained if all of the jobs run in the same time as the longest job running alone. While this might seem very unlikely to those who are familiar with traditional timeshared machines, it is possible for a multithreaded machine to devote the *unused* resources of one job to other jobs and thus complete all the jobs in the same time as one running alone.

With these preliminaries, symbiosis is defined as

$$\text{Symbiosis} = (TR - 1) / (TM - 1).$$

This gives a positive value of at most 1 if throughput is increased by running the jobs simultaneously and gives a zero or negative value otherwise.

Empirical Observations

To quantify the effect of symbiosis on a multithreaded architecture, the five NPB 2.3-serial kernels were run as simultaneous jobs for each of 15 pairings (including self pairings) on the Tera MTA. These are the same kernels used in the previously described case study, except that their size was Class W, which is smaller than the Class A and B sizes considered before.

Figure I-1 shows the results of running all 15 pairs of tuned kernels on a single MTA processor. In most cases the symbiosis is positive and sizable. In other words, the MTA is able to devote resources, such as streams, that are unused by one job to another, thus increasing throughput.

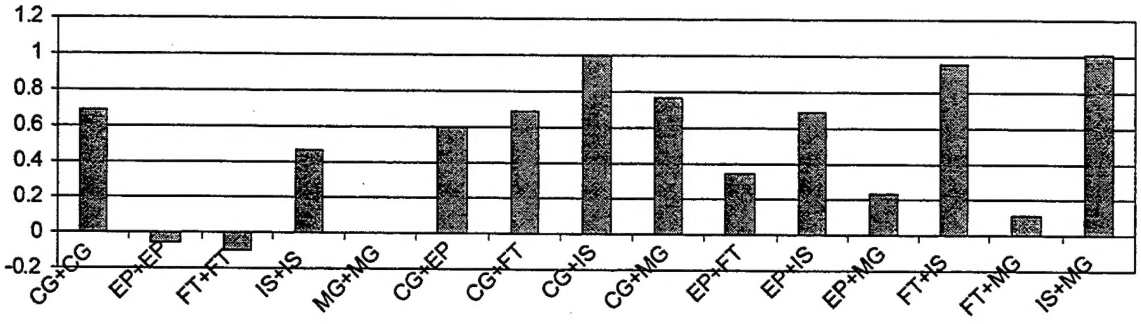


Figure I-1. Symbiosis of tuned NPB 2.3-serial kernels on a single MTA processor

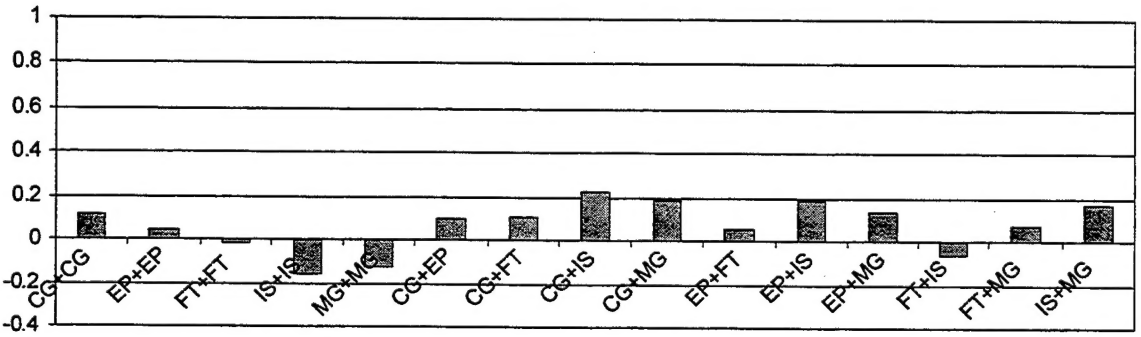


Figure I-2. Symbiosis of tuned "long-run" NPB 2.3-serial kernels on a single MTA processor

Much of the postivie symbiosis observed is due to overlap of serial sections in one job with parallel (or serial) sections of the other job. Even though all five kernels have well-tuned parallel sections that are timed in the usual benchmark tests, there are substantial serial sections associated with input/output and initialization that are not normally timed. These serial sections are included in the throughput analysis here and lead to positive symbiosis. Thus the tests corresponding to the results in Figure I-1 are not necessarily representative of the more demanding and typical case in which applications with relatively short serial sections run simultaneously.

To simulate the more demanding case, the five kernels were modified to have parallel sections that run longer, which made the serial sections proportionately smaller. Figure I-2 shows the results of running all 15 pairs of these "long-run" kernels. The benefits of symbiosis are much reduced, but still positive on balance.

Symbiotic Job Scheduling

Symbiosis can be enhanced by appropriately scheduling jobs. The tests reported here have already resulted in improvements to the current MTA job scheduler.

Most of the cases of negative symbiosis seen in the reported tests were due to a lack of dynamic resource allocation in the operating system. If two jobs started at the same time, one might grab a lot of streams. The second one would get whatever streams were left over. If the first job subsequently released its streams, the second job would not be notified of this and would proceed with fewer streams than were available. This "stream throttling" was brought to the attention of Tera

staff who came up with several dynamic resource allocation features in the current operating system. These, to a large degree, have eliminated cases of negative symbiosis on the MTA.

Reference

I-1. A. Snavely, N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "Explorations in Symbiosis on Two Multithreaded Architectures," *Proceedings of Workshop on Multi-Threaded Execution, Architecture, and Compilers,* Orlando FL (January 1999).